

Stołeczny Ośrodek
Elektronicznej
Techniki Obliczeniowej

INFORMATYKA
mikrokomputerowa

JAN RUSZCZYC

ASEMBLER 6502

ATARI

Warszawa 1987

SOETO 20 LAT

Jest to pierwszy polski podręcznik programowania w asemblerze 6502 przydatny wszystkim, których komputery wyposażone są w ten lub pokrewny mikroprocesor. Należą do nich szczególnie popularne w Polsce komputery Atari 800XL, 65XE i 130XE, Commodore C64, a także Apple, Acorn, Laser i inne.

Asembler 6502 pozwala tworzyć programy szczególnie sprawne i efektywne. Książka umożliwia poznanie tego języka w stopniu wystarczającym do samodzielnego programowania. Zawiera obszerną dokumentację pomocną doświadczonym programistom.

Autor zebrał i przystępnie przedstawił informacje rozproszone w trudno dostępnych aktualnych publikacjach, głównie anglojęzycznych. Lekturę ułatwiają liczne przykłady programów, ćwiczenia, a także rysunki.

JAN RUSZCZYC

A S E M B L E R   6 5 0 2

Druk ZRiWDB Warszawa ul. Królewska 27 Format A-5 Nakład 3300

Zam. Nr 47 1988 II 2 Ark. Dr. 15,875 K-30

P R Z E D M O W A

Książka ta ma na celu dostarczenie informacji i wskazówek umożliwiających samodzielne programowanie w asemblerze i języku maszynowym na komputerach wyposażonych w mikroprocesory 6502 i pokrewne.

Należą do nich w szczególności popularne w świecie i w Polsce mikrokomputery Commodore C64 i C128, oraz kolejne wersje cenionego komputera osobistego Apple: II, IIe, IIc, II+, III, a także nowe mikrokomputery Acorn Master, Acorn Compact, Laser 3000 i inne. W swej zasadniczej treści książka niniejsza może być pomocna użytkownikom w s z y s t k i c h tych komputerów, ponieważ dotyczy wspólnego dla nich języka.

Głównym "warsztatem pracy" przy pisaniu były szeroko u nas znane 8-bitowe komputery Atari 800XL i 130XE, które w dalszym tekście nazywane będą w skrócie Atari. Na ile pozwalały rozmiary publikacji i założony stopień ogólności, właściwościom tych komputerów poświęciłem dodatkową uwagę, zwłaszcza w rozbudowanym rozdziale 9.

Równolegle z rozwojem języków wysokiego poziomu widoczne jest współcześnie niesłabnące masowe zainteresowanie językami maszynowymi czyli tymi, w których komputery w rzeczywistości pracują. Wynika ono przede wszystkim z praktycznie potwierdzonej wysokiej efektywności kodu maszynowego. Co roku ukazuje się w różnych krajach dziesiątki publikacji poświęconych asemblerowi i językowi maszynowemu różnych mikroprocesorów, w tym 6502. Z chęci złagodzenia luk, jakie u nas występują jeszcze pod tym względem, zrodziła się niniejsza książka.

Jest ona w założeniu przewodnikiem p o p u l a r n y m, przeznaczonym przede wszystkim dla tych, którzy w sztuce programowania nie nabyli jeszcze doświadczenia. Równocześnie informacje zgromadzone w książce, w tym w jej rozbudowanych aneksach, mogą być użyteczne dla zaawansowanych programistów. Zakłada się, że Czytelnik zna przynajmniej najpopularniejszy z

języków wysokiego poziomu:Basic, który posłużył także jako pomoc w wyjaśnianiu wielu kwestii.

Jak czytać książkę? Zamierzeniem przy jej pisaniu było zaznajomienie Czytelnika z p o d s t a w a m i programowania w asemblerze oraz jego specyficznymi właściwościami. Znalazło to wyraz w treści rozdziałów wyjaśniających zasady pracy komputera, sposoby kodowania i przetwarzania informacji, szczególne cechy komputerowej arytmetyki. Pomoc w opanowaniu tych i innych kwestii stanowić powinny liczne przykłady, a także ćwiczenia do samodzielnego wykonania. Rozwiązania ćwiczeń oznaczonych "x" podano po ostatnim rozdziale. Należy unikać przechodzenia do następnych partii książki bez przyswojenia wcześniejszych.

Wszystkie rozkazy i tryby adresowania 6502 zostały wyjaśnione z pomocą przykładów. Aneksy zawierają opis rozkazów i ich najważniejszych zastosowań oraz wiele innych informacji pomocniczych. Omówiono sposoby pisania i analizy programów w asemblerze.

W bibliografii wymieniono pozycje pomocne w dalszym poszerzaniu wiedzy. Odwołania do nich w tekście oznaczone są numerami w nawiasach kwadratowych. Szczególne uznanie pragnę przy tym wyrazić dla prac W. Douglasa Maurera [3] i Rodneya Zaksa [4], które w poważnej mierze inspirowały mnie w konstruowaniu układu i zawartości tej książki.

J.R.

Rozdział 1

WIADOMOŚCI WPROWADZAJĄCE

1.1. Trzy typy języków - podobieństwa i różnice

Rozwój języków programowania nieodłącznie towarzyszący postępowi w technice komputerowej doprowadził do wytworzenia się trzech rodzajów, a zarazem niejako trzech pięter tych języków: maszynowych, symbolicznych, w tym asemblerów, oraz języków wysokiego poziomu. Tę samą klasyfikację opatruje się także innymi nazwami określając trzy rodzaje języków jako: maszynowe, zorientowane maszynowo i zorientowane problemowo.

Mimo istotnych różnic wszystkie one mają szereg ważnych cech wspólnych. Wszystkie są środkami porozumiewania się człowieka z komputerem i muszą być podporządkowane regułom tego specyficznego dialogu, odmiennego niż toczony między ludźmi, a zwłaszcza regule jednoznaczności i dokładności wszystkich pojęć.

Wspólną cechą języków programowania jest to, że pozwalają człowiekowi wyznaczać komputerowi zadania, wywoływać jego zautomatyzowane działanie i uzyskiwać użyteczne skutki, wyniki. Wymaga to, by istniały obiekty takiego działania, które nazywamy danymi, oraz by istniał skończony i dobrze określony zbiór poleceń dla komputera zwanych instrukcjami.

Sztuka programowania polega na zbudowaniu poprawnego ciągu instrukcji, które wykonywane przez komputer kolejno według ustalonego porządku doprowadzą do zrealizowania zadania. Ów ciąg instrukcji nosi nazwę programu.

Wspólną cechą wszystkich języków jest to, że aby program mógł być wykonany, zarówno on, jak i przetwarzane przezeń dane muszą znaleźć się w pamięci komputera.

Czym natomiast różnią się wspomniane trzy piętra języków programowania? Tym przede wszystkim, że im wyżej wstępujemy,

tym bardziej oddalamy się od konkretnego komputera i jego specyficznych cech i możliwości, a jednocześnie zbliżamy się do zasobu pojęć i słów człowieka i jego mowy. Im wyższe piętro, tym łatwiej jest programować, ale też na ogół tym trudniej jest przełożyć zadania na jedyny język, jaki w rzeczywistości "rozumie" komputer, tzn. język maszynowy.

1.1.1 Język maszynowy

Czym jest język maszynowy (JM) zwany również kodem maszynowym (ang. machine code) ? Jest to podstawowy język komputera. Jego instrukcje, które nazywać będziemy dla wyróżnienia r o z k a z a m i, są zakodowane w liczbach i kierowane wprost do jednostki przetwarzającej dane w komputerze: p r o c e s o r a. Program w JM - to uporządkowany ciąg rozkazów.

W komputerze można zastosować więcej niż jeden procesor. Np. w Atari jest drugi procesor ANTIC do obsługi obrazu. Zawsze przy tym jeden procesor pełni rolę jednostki centralnej (ang. central processing unit - CPU). Do takich właśnie funkcji przeznaczony jest mikroprocesor 6502. Przedrostek "mikro" oznacza urządzenie wytwarzane w postaci układu scalonego o wysokiej skali integracji.

Każdy procesor realizuje rozkazy należące do właściwego dlań zbioru czyli listy rozkazów (ang. instruction list) w ustalonych trybach adresowania (ang. adressing modes). Jest zatem tyle języków maszynowych, ile mikroprocesorów o odmiennych listach rozkazów.

Programowanie w JM jest pracochłonne. Trudno jest pisać programy złożone z samych liczb, a jeszcze trudniej je odczytać bez specjalnych narzędzi pomocniczych. Zagadką dla laika może być, na przykład, dlaczego w jednym wypadku liczba dziesiętna 32 oznacza rozkaz skoku do podprogramu, analogiczny do GOSUB w Basicu, innym razem jest natomiast, powiedzmy, składnikiem w dodawaniu. Jak mikroprocesor potrafi to odróżnić, wyjaśnimy później.

Istotne ograniczenia wynikają także ze ścisłego uzależnienia JM od listy rozkazów danego procesora. Uniemożliwia to przenoszenie tekstów programów na inne komputery z wyjątkiem

sytuacji, gdy lista rozkazów jednego mikroprocesora jest taka sama bądź stanowi rozszerzenie listy drugiego, czego przykładem są 6502, 6510 i 65C02 lub 8080 i Z80.

1.1.2 Asembler

Aby usprawnić programowanie w JM, stworzono język pomocniczy: asembler, zaliczany do kategorii języków symbolicznych. Umożliwia on programiście zastąpienie liczbowych kodów rozkazów i danych łatwiejszymi do zapamiętania nazwami symbolicznymi. Dostarcza także wielu innych ułatwień. Nie zmienia przy tym zasadniczej struktury programów w JM: jednemu rozkazowi asemblera ściśle odpowiada jeden rozkaz maszynowy. Dlatego mówi się, że asembler jest zorientowany maszynowo.

Asembler bardzo znacznie ułatwia i przyspiesza tworzenie kodu maszynowego. Wyłania się jednak istotny problem, odnoszący się również do języków wysokiego poziomu: konieczność przełożenia translacji programu napisanego w języku asemblera na JM. Czynność tę wykonuje specjalny program noszący również nazwę asemblera.

Program napisany w języku asemblera zwany jest programem źródłowym (ang. source code), natomiast efektem asemblowania jest powstanie programu wynikowego (ang. object code) w języku maszynowym. Wyróżniliśmy poprzednio pojęcia: program i dane. Warto zauważyć, że dla programu wykonującego asemblowanie tekst programu źródłowego ma w całości charakter danych do przetworzenia. Wskazuje to na pewną względność pojęcia: dane.

Jest tylko jeden język maszynowy danego mikroprocesora, np. 6502, natomiast asemblerów do niego można stworzyć wiele. Dla każdego mikroprocesora powstają z reguły asemblery o rozmaitym stopniu złożoności - od prostych do wyposażonych w liczne narzędzia pomocnicze. Najwięcej ułatwień zapewniają makroasemblery, umożliwiające m.in. zastępowanie ciągu wielu rozkazów tzw. makrorozkazem.

Dwa najważniejsze udogodnienia w asemblerze - to możliwość stosowania mnemoników kodów operacji i etykiet.

Kod mnemoniczny czyli mnemonik (od greckiego słowa: mnemos - pamięć) - to skrót literowy zastępujący w asemblerze kod operacji czyli tę część rozkazu, która określa zadanie i sposób jego wykonania (ang. opcode). Wytwórcy procesorów z reguły proponują zestaw nazw mnemonicznych dla danej listy rozkazów. Firma MOS Technology, pierwszy wytwórca mikroprocesora 6502, zaproponowała udany zestaw mnemoników, który się powszechnie przyjął i znacznie ułatwia m.in. porozumiewanie się użytkowników rozmaitych mikrokomputerów wyposażonych w 6502, jak Apple, Atari i Commodore.

Wszystkie mnemoniki dla 6502 są trzyliterowe i stanowią skróty angielskich określeń wykonywanych czynności. Na przykład, mnemonik "LDA" - to skrót polecenia: "load accumulator" czyli "załaduj akumulator".

Etykiety - to również mnemoniczne nazwy, którymi programista może po uprzednim zdefiniowaniu zastąpić w programie rozmaite dane liczbowe, a także związać z etykietami adresy w samym programie. O stosowaniu etykiet powiemy szerzej w punkcie 1.6.2.

Asembler umożliwia posługiwanie się przy tworzeniu programu pomocniczymi obliczeniami. Jedynie ich wyniki umieszczane są w kodzie wynikowym.

Dzięki tym i innym ułatwieniom asembler w poważnej mierze wyręcza programistę w mozolnej i uciążliwej pracy obliczeniowej, pozwalając mu skoncentrować główną uwagę na treści programu i jego strukturze.

1.1.3 Języki wysokiego poziomu

Dopiero w około 10 lat po skonstruowaniu pierwszych komputerów rozpoczęło się tworzenie trzeciego z wspomnianych na wstępie pięter w hierarchii języków programowania. W r. 1954 opracowano, a w r. 1956 zastosowano na komputerze IBM nowy język programowania, który otrzymał nazwę Fortran. Była ona skrótem słów: FORmula TRANslator - tłumacz formuł. Zgodnie ze swą nazwą język ten powstał w celu uproszczenia obliczeń matematycznych i naukowych. Rolę taką pełni również dziś.

Był to pierwszy język programowania wysokiego poziomu.

Po nim przyszły następne, a ich łączną liczbę szacuje się dziś na około 5000, z czego tylko około 10 zdobyło szeroką popularność. Mówi się o nich, że są maszynowo niezależne, choć takiego ideału w pełni osiągnąć się nie udaje. Języki te pozwoliły abstrakcyjnymi pojęciami, takimi jak stałe, zmienne, tablice, rekordy i wiele innych, zastąpić niezbędne w języku maszynowym odwoływanie się do adresów w pamięci i rejestrów mikroprocesora.

Programowanie zbliżono do mowy ludzkiej. Instrukcje przybrały postać poleceń wyrażanych słowami języka - przede wszystkim angielskiego, który zadomowił się w słownictwie informatycznym na całym świecie. Wiele instrukcji Basicu i innych języków programowania - to po prostu słowa angielskie określające czynność lub zjawisko: print - drukuj, read - czytaj, data - dane, go to - idź do, go sub - idź pod, sound - dźwięk, graphics - grafika itd.

Ważnym udogodnieniem stało się to, że gdy pojedyncze rozkazy JM realizują się w większości wąskie, cząstkowe zadania, w językach wysokiego poziomu pojedyńcza instrukcja zastępuje dziesiątki, a nawet setki rozkazów maszynowych. Struktura tych języków umożliwiła rozbudowę systemu kontroli błędów utrudnionej w JM.

Języki wysokiego poziomu znacznie uprościły i przyspieszyły programowanie stając się ważnym ekonomicznie czynnikiem obniżenia jego kosztów. Ich rozwój wytworzył z jednej strony języki przeznaczone do wysoce specjalistycznych zadań, jak sterowanie robotami, z drugiej zaś dominujące stało się dążenie do tworzenia efektywnych języków ogólnego przeznaczenia, takich jak, z reguły dostępne na mikrokomputerach, Basic, Pascal, Forth, język C i Logo.

Zróżnicowanie cech poszczególnych języków zapewnia szerokie możliwości doboru takich, które pozwalają najskuteczniej realizować konkretne zadania lub najbardziej odpowiadają upodobaniom programisty.

Języki wysokiego poziomu postawiły zarazem na porządku dziennym problem, który sygnalizowaliśmy już przy asemblerze: konieczność translacji programu źródłowego na kod wynikowy w

JM. Zadanie takie wykonuje program zwany translatorem (ang. translator). Dwie podstawowe formy translacji noszą nazwę interpretacji i kompilacji.

Interpretator (ang. interpreter) tłumaczy program źródłowy instrukcja po instrukcji i natychmiast go wykonuje. Stosuje się go w językach dialogowych czyli konwersacyjnych, takich jak Basic czy Logo. Program źródłowy, którego listing możemy czytać i dowolnie zmieniać, tłumaczony jest i wykonywany za każdym razem od nowa.

Inaczej działa kompilator (ang. compiler) . Podobnie jak asembler dokonuje dwu, a niekiedy trzyetapowego tłumaczenia programu źródłowego i dopiero po ukończeniu tej pracy program wynikowy jest gotów do wykonania.

W obu formach translacji użytkownik za ułatwienia w programowaniu musi zapłacić niemałą cenę na dwóch przede wszystkim odcinkach: czasu wykonania oraz rozmiaru pamięci dostępnej dla programów i danych.

Wydłużenie się czasu wykonania jest szczególnie odczuwalne w językach dialogowych, mniej w kompilowanych. W obu wypadkach strata czasu w poważnej mierze zależy od jakości translatorów. Wiedzą o tym dobrze użytkownicy rozmaitych wersji Basicu na te same komputery. Ogólnie jednak nigdy nie udaje się przy kompilacji czy interpretacji osiągnąć takiej szybkości wykonania, jaką zapewnia dobry program opracowany w asemblerze.

Druga niedogodność - pamięciochłonność języków wysokiego poziomu - jest bardziej odczuwalna w przypadku mikrokomputerów niż dużych maszyn cyfrowych. Na łączne zaabsorbowanie pamięci rzutuje nie tyle długość samych programów, co przede wszystkim konieczność rozmieszczenia w pamięci translatorów, a także różnorodnych pomocniczych tablic, których wymagają języki interpretowane, jak i kompilowane.

Np. znajdujący się w pamięci stałej Atari interpretator Basicu zajmuje 8 kilobajtów oraz wykorzystuje dodatkowo kilka sporych stref pamięci komputera. Miejsce to zwalnia się, gdy programujemy w asemblerze.

Bardzo rozrzutne pod względem wykorzystywania pamięci jest Logo. W niemniejszym stopniu dotyczy to Lispu - języka przetwarzania pamięciochłonnych struktur spisowych.

Im większe są możliwości języka i dłuższa jego lista instrukcji, tym na ogół bardziej skomplikowane i rozbudowane stają się translatory. Niektóre kompilatory są tak obszerne, że zajmują niemal całą pamięć i wymagają kompilowania programu wynikowego na dyskietkę lub inny zewnętrzny nośnik pamięci. Są również języki o tak rozbudowanych środkach, że na mikrokomputerach możliwe staje się zrealizowanie tylko ich podzbiorów czyli wersji okrojonych. Istnieją także korzystne wyjątki. Na przykład, Forth, którego translator pełni równocześnie funkcje interpretatora i kompilatora, pozwala tworzyć programy bardzo zwięzłe i odznaczające się dużą szybkością wykonania.

W przypadku języków kompilowanych, takich jak C, Pascal i wiele innych, czy też kompilatorów programów napisanych w językach dialogowych, np. kompilatorów z Basicu, sporo miejsca zajmują specjalne moduły programowe niezbędne do wykonania programu. Są one dołączane do programów wynikowych lub muszą być przed ich wykonaniem wprowadzone do pamięci.

W każdym wypadku, gdy stoimy przed wyborem języka programowania, wyłania się problem zbilansowania plusów i minusów. Nie można przy tym przeoczyć zalet języków wysokiego poziomu: pozwalają one programować szybciej i sprawniej, ułatwiają przenoszenie programów na komputery o innych procesorach, a także wykrywanie i usuwanie błędów. Pamiętajmy zarazem: nie ma języków lepszych i gorszych i nie ma takiego, którym miałby same tylko zalety. Wybór języka zależy przede wszystkim od charakteru realizowanych zadań.

Ćwiczenia

1. Jaka jest podstawowa różnica między interpretatorem a kompilatorem?

2. Co to jest program źródłowy, a co program wynikowy?

x 3. Czym różni się etykieta od mnemoniku?

1.2 Kiedy i dlaczego asembler?

Praktyka potwierdza, że między językami wysokiego poziomu a kodem maszynowym i służącym do jego tworzenia asemblerem nie ma przeciwieństwa, że możliwe i celowe jest wiązanie ich i kojarzenie. Kompilatory języków wysokiego poziomu z reguły przewidują włączanie do programów wstawek w języku maszynowym. Również języki interpretowane, w tym Basic, umożliwiają posługiwanie się kodem maszynowym, a programiści z reguły z tego korzystają. Forth, sam ściśle powiązany z komputerem, ma ponadto własny bardzo wygodny asembler do definiowania fragmentów w tym języku. Znamienne jest również, że autorzy popularnego ostatnio języka C wprowadzili bezpośrednio do arsenału jego instrukcji szereg środków języka maszynowego, takich jak operacje na bitach oraz zwiększanie i zmniejszanie wartości zmiennych o 1.

Kiedy celowe jest przejście na pracę w asemblerze i tworzenie kodu maszynowego? Odpowiedź wynika wprost z tego, co zostało wcześniej powiedziane:

- gdy niezbędne jest maksymalne skrócenie czasu wykonania,
- gdy trzeba oszczędzać pamięć.

Określa to następujące główne dziedziny zastosowań języka maszynowego, a zatem również programowania w asemblerze:

1. Obsługa dróg łączności z urządzeniami zewnętrznymi wymagającymi szybkiego przesyłu danych.
2. Praca w tzw. czasie rzeczywistym, np. kontrola procesów produkcyjnych z pomocą czujników i urządzeń pomiarowych.
3. Obsługa obrazu graficznego (ang. graphics display), co z reguły wymaga operowania niewielkimi elementami obrazu, pikselami (ang. pixel - picture element) odpowiadającymi często bezpośrednio bitom. Także programowanie dźwięku.

Za poznawaniem JM i asemblera przemawiają również inne względy. Języki te pozwalają lepiej zrozumieć architekturę mikrokomputera i wewnętrzne mechanizmy jego działania. Dzięki ich poznaniu możemy głębiej wniknąć w strukturę języków wysokiego poziomu i efektywniej w nich programować. Stosowanie

wstawek JM w programach napisanych w języku wysokiego poziomu może, zwłaszcza w przypadku dużych obliczeń, znacznie skrócić czas wykonania. Znajomość JM i asemblera otwiera drogę do skutecznej analizy i ewentualnych zmian w programach napisanych przez innych, np. w grach.

Nie będzie zatem przesadą, gdy do trzech wyliczonych uprzednio punktów dodamy czwarty: rolę poznawania języka niskiego poziomu w podnoszeniu tak potrzebnej współcześnie kultury informatycznej społeczeństwa, w tym rzeszy garnącej się do komputerów młodzieży. Sądzę, że jest to język dla niej po prostu ciekawy.

1.3 Narzędzia skutecznego programowania

Podstawowe metody programowania w JM i asemblerze są zbieżne ze stosowanymi w językach wysokiego poziomu. Istnieje jednak parę dość istotnych różnic. Jesteśmy jakby bliżej maszyny, pomocne stają się zatem informacje o tym, jak działa. Nie chodzi o wiedzę ściśle elektroniczną, lecz o znajomość logiczno-matematycznych zasad działania komputera. Dlatego też za celowe uznać należy poszerzenie wiedzy programującego w JM w następujących dziedzinach:

- wewnętrzna organizacja komputera, a zwłaszcza procesora;
- sposób, w jaki reprezentowane są w komputerze informacje;
- stosowane kody, w tym zwłaszcza kod liczb binarnych;
- posługiwanie się liczbami binarnymi i szesnastkowymi;
- sposób wykonywania przez CPU obliczeń arytmetycznych i logicznych.

Z konieczności skrótowy przegląd tych kwestii jest treścią następnych partii książki.

1.4 Forma przedstawienia informacji w komputerze

Programowanie w JM i asemblerze wymaga znajomości sposobu, w jaki przedstawiane i przetwarzane są w komputerze wszelkie informacje. Znajduje w nim zastosowanie wiele sposobów

kodowania programów i danych, główny jednak polega na wykorzystaniu do tego liczb binarnych. W kodzie tym, analogicznie do pozycyjnego dwójkowego czyli binarnego układu liczbowego, wszelkie wartości wyrażane są z pomocą jedynie dwóch cyfr: 0 i 1.

Dlaczego wybrano taką właśnie formę reprezentacji danych, z pozoru niewygodną ze względu na nasze przezwyczajenia do posługiwania się układem dziesiętnym (dec.)? Kod binarny znalazł zastosowanie przede wszystkim dlatego, że najtrafniej odzwierciedla w postaci liczb to, iż elektryczne układy przełączające i ładunki mogą znajdować się tylko w dwóch stanach: włączony lub wyłączony, naładowany lub rozładowany. W operacjach logicznych, na których opiera się działanie komputera, występują także dwa stany: prawda i fałsz, co również można wyrazić dwiema cyframi: 1 i 0.

Najmniejszą jednostką informacji jest bit (skrót angielskiego określenia BInary digiT) - cyfra binarna, który może przybierać dwie wartości: 0 i 1. W JM i asemblerze nieustannie występuje konieczność sprawdzania i zmiany wartości bitów.

Osiem bitów zgrupowanych jest w większą jednostkę: bajt (ang. byte). Jest to w komputerach opartych na 6502, podobnie jak w innych komputerach 8-bitowych, podstawowa jednostka przechowywania danych w pamięci i ich przetwarzania przez mikroprocesor. Nazywa się ją również słowem mikroprocesora. Jednakże w praktyce programowania w asemblerze pojęcie słowa (ang. word) używa się najczęściej do określenia liczb i jednostek pamięci o rozmiarze dwóch bajtów. Wielkość ta jest wykorzystywana przez mikroprocesor w obliczaniu adresów w pamięci. Adres - to niejako numer bajtowej komórki pamięci, którym posługuje się mikroprocesor w celu jej odnalezienia. Należy pamiętać, że 6502 odczytuje adresy 16-bitowe w odwróconej kolejności składających się na nie bajtów. W pierwszym bajcie mieści się 8 mniej znaczących bitów liczby binarnej (ang. least significant byte - LSB), w drugim - bity bardziej znaczące (most significant byte - MSB). W takiej też kolejności umieszcza je asembler w programie wynikowym w części rozkazu zwanej operandem lub argumentem.

Wymieńmy dwie większe miary stosowane w opisie pracy komputera i określaniu rozmiarów pamięci. Kilobajt (KB) liczy 1024 bajty, a megabajt (MB) 1024 KB czyli 1048576 bajtów.

Ćwiczenia

1. Co to jest: a/ bit, b/ bajt, c/ LSB, d/ MSB, e/ KB, f/ MB?

1.5 Liczby binarne i szesnastkowe

Binarny zapis liczb jest niewygodny do czytania i trudno jest go zrozumieć. Przy pisaniu programów w JM i asemblerze szerokie zastosowanie znajduje inny zapis: w szesnastkowym czyli heksadecymalnym (hex) pozycyjnym układzie liczbowym. Układ ten posługuje się zgodnie z nazwą 16 cyframi, jak dziesiętny - dziesięcioma. Liczbę cyfr nazywamy też p o d s t a w ą pozycyjnego układu liczbowego. Brakujące znaki cyfr powyżej 9 zastępuje się literami alfabetu. Układ szesnastkowy posługuje się zatem cyframi: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E i F. Tak więc np. A hex = 10 dec, a F hex = 15 dec.

Zaletą układu jest to, że wartość binarna mieszcząca się w bajcie może w nim być przedstawiona z pomocą dwóch cyfr hex, z których każda reprezentuje połowę bajtu. Jest to szczególnie wygodne wtedy, gdy liczba 16-bitowa jest przedstawiona w pamięci w odwróconej kolejności bajtów. W układzie dziesiątkowym ustalenia jej wartości wymaga wówczas żmudnych obliczeń, w hex odczytuje się ją wprost.

W układzie dziesiętnym 10 jest dziesięć razy większe niż jeden, a 100 dziesięć razy większe niż 10. W układzie dwójkowym 10 jest d w a razy większe niż 1, a 100 dwa razy większe niż 10. Dlatego numerując kolejno bity od skrajnego prawego otrzymujemy następujące wartości dziesiętne odpowiadające kolejnym pozycjom bitów w przykładowej liczbie binarnej 10110101:

Numer bitu:	b7	b6	b5	b4	b3	b2	b1	b0
Liczba binarna:	1	0	1	1	0	1	0	1
Wartości bitów:	128	64	32	16	8	4	2	1

Aby obliczyć dziesiętną wartość liczby, trzeba zsumować te wartości z ostatniego rzędu, które odpowiadają jedynkom w liczbie binarnej: 128+32+16+4+1=181.

Przeliczenie (konwersja) liczby binarnej na hex jest znacznie prostsze. Dzielimy liczbę, poczynając od p r a w e j skrajnej cyfry na odcinki po 4 bity, po czym ustalamy wartości tych czwórek i zapisujemy je w hex, jako kolejne cyfry.

Binarnie:	1011	0101
Dziesiętnie:	11	5
Szesnastkowo:	B	5

B5 - to przeliczona na hex wartość naszej liczby.

O przeliczaniu liczb z jednego układu na inny powiemy szerzej w rozdziale 3.

Ćwiczenia

x 1. Podaj dziesiętne wartości następujących liczb binarnych: a/ 11111111, b/ 1000001, c/ 11000000, d/ 1001111.

2. Przelicz te same wartości na hex.

3. Liczby binarne w dwóch kolejnych bajtach mają poniższe wartości hex. Napisz te liczby w postaci binarnej przyjmując, że ich MSB umieszczone są jako drugie. a/ 0F 00 b/ 00 F0 c/ 01 10 x d/ C0 DD.

1.6 Od algorytmu do programu

Postulat ładu w większym niż innych języków stopniu dotyczy asemblera i JM. Przejrzystość programu, podzielenie go na moduły i zapewnienie właściwych powiązań między nimi, wyraźne wyodrębnienie poszczególnych zadań cząstkowych i zestawienie ich w jedną całość - to zadania tym ważniejsze, że programy w asemblerze są mniej czytelne niż w innych językach, nie mówiąc już o kodzie maszynowym, którego bez specjalnych narzędzi w praktyce niesposób odczytać.

Pierwszym krokiem jest jasne sformułowanie zadania, które ma wykonać komputer. Wiele niepowodzeń wynika z tego, że niedość jasno określiliśmy cel, który chcemy osiągnąć. Kolejny niezbędny krok - to opracowanie a l g o r y t m u czyli poprawnie określonego zbioru zasad i czynności niezbędnych do wykonania zadania. Algorytm nie stanowi jeszcze programu, lecz jego szkielet, konstrukcję nośną. Najczęściej stosuje się dwie formy zapisu algorytmu:

- słowną, polegającą na opisaniu czynności i wskazaniu kolejności ich wykonania. Jej wariantem jest opisanie algorytmu w uproszczonej wersji jednego z języków programowania, np. Algolu lub Pascalu;
- graficzną, noszącą nazwę schematu blokowego lub sieci działań (ang. flowchart).

Doceniać warto drugą z tych metod. Zacytujmy żartobliwą uwagę Rodneya Zaksa [5] : "Zaobserwowano, że zapewne 10 proc. zbiorowości programujących potrafi pomyślnie napisać program nie posługując się siecią działań. Niestety, zaobserwowano również, że 90 proc. tej zbiorowości wierzy, iż należy do owych 10 procent! Wynik: 80 procent programów ujawnia swą błędność już przy pierwszym uruchomieniu". Mimo to, początkujący programiści rzadko dostrzegają konieczność narysowania sieci działań.

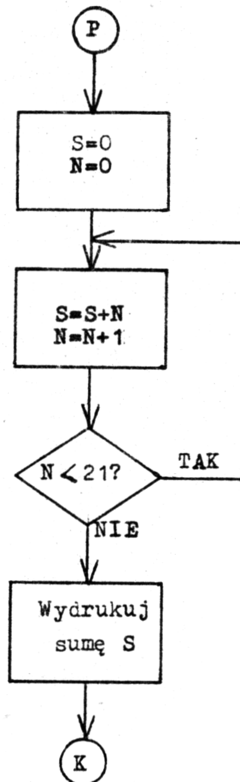
Rozpatrzmy przykład prostego algorytmu. Zadanie brzmi: "Obliczyć sumę liczb całkowitych w przedziale od 1 do 20". Algorytm w postaci słownej może brzmieć następująco:

1. Przypisać sumie S wartość 0
2. Przypisać liczbie L wartość 0
3. Przypisać S wartość S+L
4. Zwiększyć L o 1
5. Sprawdzić, czy L jest mniejsze od 21
6. Jeżeli tak, wrócić do punktu 3
7. Wydrukować wynik.

Pierwsze dwa punkty algorytmu zawierają czynność inicjalizacji zmiennych, czyli przypisania im wartości początkowych. Punkty 3-6 stanowią trzon algorytmu, przy czym w punkcie 5 wykonuje się ważną czynność sprawdzenia warunku, a punkt 6

oznacza uruchomienie powtórzeń czyli pętli, gdy warunek jest spełniony.

Rys. 1.1 przedstawia ten sam algorytm w postaci graficznej.



Rys. 1.1 Schemat blokowy algorytmu sumy 20 liczb.

Punkt początkowy i końcowy zwykło się oznaczać kółkami z napisami P i K. Kolejne czynności umieszcza się w prostokątach, warunki w rombach, przy czym wychodzące z nich linie ze strzałkami wskazują dalszy bieg obliczeń w przypadku spełnienia bądź niespełnienia warunku.

Gdy algorytm został opracowany, można przystąpić do tworzenia programu, to znaczy do przekładania go na jeden z języków programowania.

W Basicu program można napisać prosto:

```
10 S=0:L=0
20 S=S+L:L=L+1:IF L<21 THEN 20
30 ? S:END
```

A jak ten program przedstawić w asemblerze? Zanim to uczynimy, poznajmy nieco lepiej zasady programowania w tym języku.

Ćwiczenia

x 1. Czym różni się algorytm od programu?

1.7 Jak zapisać program w asemblerze?

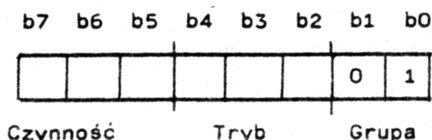
1.7.1 Budowa rozkazu języka maszynowego

Struktura zapisu programu w asemblerze odwzorowuje w zasadzie zapis w języku maszynowym, który jest z kolei sekwencją rozkazów. Jak zbudowany jest pojedyńczy rozkaz JM?

Może on zajmować jeden, dwa lub trzy bajty. Zawsze w pierwszym, często jedynym, bajcie mieści się kod operacji (ang. opcode) . Jest to część mimo niewielkiego rozmiaru bardzo pojemna informacyjnie. Osiem bitów kodu operacji podzielonych jest na pola określające grupę, do której należy rozkaz, czynność, którą wykonuje, oraz tryb adresowania. Dzięki takiej strukturze bajt kodu operacji wskazuje:

- rodzaj wykonywanej operacji;
- sposób znajdowania jednego lub dwóch argumentów czyli danych wykorzystywanych w operacji;
- miejsce przesłania wyniku;
- długość rozkazu w bajtach pozwalającą ustalić adres następnego rozkazu w programie znajdującym się w pamięci.

Np. wspomniany już rozkaz LDA (załaduj akumulator) należy do grupy ośmiu "najpotężniejszych" rozkazów, w której są ponadto: STA, ADC, SBC, CMP, AND, ORA i EOR. Ich kody operacji zbudowane są jak na rys. 1.2.



Rys. 1.2 Przykład budowy kodu operacji

Sygnalizujemy tu jedynie ciekawy sposób budowy kodu operacji rozkazu 6502. W aneksie A6 Czytelnik znajdzie opis budowy wszystkich kodów.

Jeżeli rozkaz jest więcej niż 1-bajtowy, to w drugim i ewentualnie trzecim bajcie mieści się operand czyli argument (ang. operand). Np. dla rozkazu LDA argumentem takim może być adres komórki pamięci, z której ma być przesłana do akumulatora 8-bitowa liczba, albo sama ta liczba.

Oto przykłady rozkazów o różnej długości zapisanych w asemblerze i w JM w układzie szesnastkowym :

Asembler	Język maszynowy
RTS	60
LDA # 80	A9 80
STA 4012	8D 12 40

Pierwszy rozkaz powoduje powrót z podprogramu (return from subroutine) i nie wymaga dodatkowych danych. Poznany już rozkaz LDA występuje tu w trybie natychmiastowym (ang. immediate addressing), co powoduje załadowanie do akumulatora wartości podanej jako operand. W asemblerze tryb ten zaznacza się znakiem "#" zwanym potocznie "hasz". Wreszcie trzeci rozkaz STA (store accumulator in memory - zapisz akumulator do pamięci) występuje tu w trybie adresowania absolutnego (ang. absolute addressing), co oznacza, że adres, pod którym ma być zapisana w pamięci wartość z akumulatora podany jest jako operand, tu dwubajtowy. Jest to adres 4012 hex. Zwróćmy uwagę na przestawienie bajtów adresu w kodzie wynikowym, o czym mowa była w punkcie 1.4.

Ćwiczenia

x 1. Czy wśród poniższych zapisów rozkazów (liczby w hex) są niepoprawne, a jeżeli tak, to które i dlaczego?

a/ LDA F0 b/ LDA #F0 c/ LDA 100 d/ LDA #100 e/ STA 0 f/ STA #0 g/ STA 1000

1.7.2 Mnemoniki i linie asemblera

W asemblerze rozkazy pisze się kolejno jeden po drugim, każdy rozkaz w nowym wierszu, zastępując mnemonikami kody operacji i wpisując po nich ewentualne operandy. Na ogół możliwe jest ich zapisywanie liczbami dziesiętnymi lub szesnastkowymi. Te ostatnie wyróżnia się zwykle w asemblerach na 6502 poprzedzając liczbę znakiem dolara "$". Asembler ułatwia wprowadzenie operandów, ponieważ liczbę dwubajtową zapisuje się w nim jako jedną wartość, a program przelicza ją na odpowiednie bajty i odwraca ich kolejność.

Najczęściej stosuje się dwa następujące sposoby zapisu: w kolejno numerowanych przez programującego liniach, analogicznie jak pisze się listing w Basicu, oraz w liniach, na których początku sam asembler wpisuje a d r e s, jaki mieć będzie w pamięci kolejny rozkaz. Wspólna dla obu tych form zapisu jest ważna zasada: jednemu rozkązowi maszynowemu odpowiada jedna linia programu źródłowego w asemblerze.

Gdy asembler zezwala na stosowanie etykiet, o czym za chwilę , wówczas wpisuje się je bezpośrednio po numerze linii i obowiązkowej jednej spacji. Natomiast mnemoniki rozkazów pisze się albo za etykietą, albo, gdy jej nie ma, najwcześniej d w i e spacje za numerem linii. Można też użyć tabulatora. Asemblery z wyjątkiem najprostszych zezwalają ponadto na wpisywanie do tekstu źródłowego komentarzy, co ma zasadnicze znaczenie dla podniesienia czytelności programów. W takim wypadku na początku linii umieszcza się średnik lub wpisuje komentarz po spacji i zwykle bez średnika za rozkazem.

Poznaliśmy jedynie najbardziej podstawowe zasady pisania programu w asemblerze, z innymi zetkniemy się w toku dalszej lektury.

W książce dla uproszczenia zapisu pomijać będziemy na ogół numerację linii. Etykiety znajdować się będą zatem na początku linii, a rozkazy nieco dalej w prawo.

1.7.3 Stosowanie etykiet

Etykiety stanowią bardzo znaczne udogodnienie. Wykorzystują się je na dwa sposoby. Pierwszy polega na tym, że na początku programu z pomocą operatora "=", w niektórych asemblerach "EQU", przypisujemy etykiecie wartość, którą zachowywać będzie odtąd w całym programie. Na przykład:

```
G = $76
```

W programie można teraz napisać rozkaz w postaci:

```
STA G
```

Asembler odczyta, że należy zawartość akumulatora przenieść do komórki o adresie G czyli 76 hex. To zastosowanie etykiety G odpowiada posługiwaniu się stałą w Basicu.

Inne, szczególnie użyteczne zastosowanie etykiety polega na wpisaniu jej na początku linii asemblera i wykorzystaniu w rozkazie skoku. Oto przykład takiej sekwencji:

		Adres	Kod
	JSR SKOK	0600	20 03 06
	BRK	0602	00
SKOK	RTS	0603	60

JSR (jump to subroutine) oznacza skok do podprogramu. Co się stanie w tym przykładzie? "JSR SKOK" spowoduje przeskok pod adres etykiety SKOK, który wyliczy asembler. Tam program znajdzie rozkaz powrotu z podprogramu: RTS. Spowoduje to powrót do rozkazu następnego po "JSR SKOK". Znajdujący się tam rozkaz BRK (break) spowoduje przerwanie wykonania programu. Po prawej stronie pokazujemy ten fragment programu (adresy i kody) przyjmując, że jego początek znajduje się pod adresem 600 hex.

W przykładach tej książki zastosujemy taką właśnie konwencję zapisu.

1.8 Wyjaśnienie asemblera z pomocą Basicu

Specyficzne metody programowania w asemblerze łatwiej jest przyswoić porównując wykonywane czynności z tymi, jakie stosujemy w Basicu, by osiągnąć analogiczny cel. W asemblerze nie ma pojęcia zmiennej. Jednakże rolę zmiennej możemy przypisać komórce pamięci. Wówczas a d r e s tej komórki można uznać za to, czym w Basicu jest n a z w a zmiennej. Przeczytajmy raz jeszcze uważnie ostatnie zdanie, bowiem przedstawioną tu analogię będziemy stale wykorzystywać w książce.

Rozpatrzmy poniższą parę rozkazów:

```
LDA H
STA G
```

Jaka czynność została wykonana? Pierwszy rozkaz spod a d r e s u oznaczonego etykietą H wprowadził do akumulatora znajdującą się tam wartość. Z kolei drugi rozkaz wartość tę umieścił pod a d r e s e m oznaczonym etykietą G. Akumulator stał się zatem pośrednikiem w przeniesieniu wartości. Dokładnie odpowiada to instrukcji przypisania w Basicu:

LET G=H lub po prostu G=H

Inny przykład:

```
LDA #25
STA G
```

Czemu w Basicu odpowiada ta para rozkazów? Należy zwrócić uwagę na wcześniej sygnalizowaną istotną różnicę w stosunku do poprzedniego przykładu. Tam "LDA H" oznaczało wpisanie do akumulatora zawartości komórki o adresie H. Tu zapis "LDA #25" oznacza coś innego: do akumulatora wpisujemy l i c z b ę 25. Znak "#" wskazuje na tryb adresowania natychmiastowego. Zapamiętajmy:

LDA 700 - to tryb absolutny, w którym wartość pobiera się spod adresu 700.

LDA #25 - to tryb natychmiastowy, w którym wartość podana jest jako operand w samym rozkazie.

Pierwszy rozkaz mieć będzie długość trzech bajtów, drugi - dwóch. W języku maszynowym "LDA" zostanie zastąpione odmien-

nymi kodami operacji. Jak widzimy, jeden mnemonik może odpowiadać rozmaitym kodom operacji.Asembler rozpoznaje to na podstawie budowy linii.

Wróćmy do naszego przykładu. Korzystając z pośrednictwa akumulatora, do komórki o adresie oznaczonym etykietą G wprowadziliśmy wartość 25. Analogiczny zapis w Basicu brzmieć będzie:

```
G=25
```

Jak wynika z tych przykładów, asembler dzięki zawartej w nim możliwości stosowania etykiet daje się w pewnej mierze upodobnić do Basicu. Korzystajmy z tej możliwości.

By pokazać to na kolejnym przykładzie, poznamy jeszcze jeden rozkaz asemblera i JM.

```
INC L
```

INC - to skrót angielskiego słowa "increment" - "zwiększenie". Czynność wykonywana przez ten rozkaz polega na zwiększeniu wartości o 1. Jest to bardzo użyteczne w pętlach. W danym wypadku zastosowany został, podobnie jak w poprzednich przykładach, tryb adresowania absolutnego. L oznacza a d r e s. Zawartość komórki pamięci o tym adresie została zwiększona o 1. Jak to wyrazić w Basicu?

```
L=L+1
```

Zastosujmy nowo poznany rozkaz w nieco bardziej rozbudowanym fragmencie programu:

```
LDA #36
STA K
STA L
INC L
LDA L
STA M
INC M
INC M
```

Wyjaśnijmy krok po kroku uzyskany efekt. Rozkazy, jak to widzimy, operują na komórkach pamięci o a d r e s a c h K, L i M. Pierwszy rozkaz wprowadził do akumulatora wartość 36,

drugi przeniósł ją do komórki o adresie K. Akumulator, pamiętajmy o tym, zawiera nadal 36. Toteż trzeci rozkaz, STA L, wpisuje do komórki o adresie L również 36. Czwarty rozkaz zwiększa zawartość tej komórki o 1. Pod adresem L jest teraz 37. Następuje LDA L. Co się w efekcie zmienia? Wartość w akumulatorze, w którym jest teraz 37. Kolejny rozkaz, STA M, sprawia, że równkomórce o adresie M zapisane zostaje 37. Dwa ostatnie rozkazy zwiększają tę wartość o 2.

Łączny efekt tej sekwencji rozkazów można zatem w Basicu przedstawić następująco:

K=36:L=37:M=L+2

Ten sam efekt można osiągnąć prościej, sześcioma rozkazami - i to na parę sposobów. Jak? Zastanówmy się nad tym sami.

Omówione przykłady wskazują, że symulowanie zmiennych Basicu nie jest w asemblerze skomplikowane. Trzeba tylko pamiętać o konieczności wcześniejszego zdefiniowania adresów K, L i M - i to tak, by były bezpieczne przed przypadkowym wtargnięciem innego czy nawet tego samego programu. Nazywamy to rezerwowaniem miejsca na dane.

LDA i STA należą w asemblerze i JM do grupy rozkazów przesłania danych. W Basicu w stwierdzeniach G=25, N=N+1 itp. wykonujemy przypisanie wartości zmiennym. Istnieje tu, jak widzimy wyraźne podobieństwo efektów.

Ćwiczenia

1. Napiszmy ciągi rozkazów odpowiadające poniższym stwierdzeniom Basicu: a/ E=F x b/ G5=G3 c/ H9=32 x d/ Z=0 x e/ E6=E4+1

2. Napiszmy w Basicu równoważniki następujących ciągów rozkazów:

a/	x b/	x c/
LDA L1	LDA #62	LDA S
STA L2	STA U	STA T
	STA W	INC T
		STA U

x 3. Ile bajtów liczą następujące rozkazy?:

a/ LDA #0 b/ STA 1000 c/ BRK d/ INC e/ RTS.

1.9 Pierwsze kroki w asemblerze

Treść tej książki można, choć brzmi to paradoksalnie, przyswajać także b e z komputera. Co więcej, przystępowanie do programowania może być efektywne dopiero wtedy, gdy pozna się dokładnie co najmniej pewną liczbę rozkazów i sposób ich stosowania.

Jest jednak naturalne, że chcielibyćmy zobaczyć, czy przedstawiane tu bądź przez nas realizowane przykłady i ćwiczenia rzeczywiście dają opisywany efekt. Wymaga to dostępu do komputera i choćby najprostszych narzędzi do pisania i wykonywania programów w asemblerze. Można, oczywiście z Basicu z pomocą POKE wstawić kolejne kody i dane programu do pamięci, a potem wykonać go z pomocą USR. Będzie o tym mowa w rozdz. 8. Pisanie w JM jest jednak uciążliwe.

Doradzam wykorzystanie do pierwszych wprawek jednego z dostępnych programów uruchamiających (ang. debugger). Na Atari najbardziej przydatny do tego celu jest BUG/65, który w parze ze świetnym makroasemblerem MAC/65 opracowanym w Optimized Systems Software (OSS) przez S.D.Lawrowa, zaspokoić może potrzeby wytrawnego programisty. Podobnie jak BUG/65 wykorzystać można "Atari Debugger". Oba programy mają w zasadzie takie same zestawy komend, a dla ćwiczącego istotne jest, że zawierają nieduży podręczny asembler. Wszystkie dane pisze się w nim w hex bez konieczności poprzedzania liczb znakiem "$". Podręczny asembler umożliwia korzystanie z niedużego zestawu etykiet, które w danym wypadku można oznaczać tylko L0, L1 ... L9. W krótkim programie tych 10 etykiet zupełnie wystarczy. Podręczny asembler nie zachowuje programu źródłowego i od razu tworzy kod wynikowy w pamięci. Nie zezwala zatem również na wpisywanie komentarzy.

Prześledźmy na niedużym przykładzie kolejne kroki pisania, sprawdzania i wykonywania programu. Na Atari wygodnym miejscem pisania niewielkich programów jest szósta strona pamięci od $600 do $6FF. Warto jednak sprawdzić, czy obszar, w którym chcemy umieścić program, jest wolny. Piszemy "D600", a BUG odpowiada:

```
0600 00 00 00 00 00 00 00 00
```

Komenda D służy do wyświetlania zawartości komórek pamięci. Gdy nie podamy adresu końcowego, wyświetli 8 komórek.

Teraz komendą Z z adresem startu wywołujemy podręczny asembler, po czym na ekranie pojawia się wpisany przez asembler pierwszy adres do wprowadzania danych:

```
Z600
0600 ■
```

Kursor (ciemny kwadrat) wskazuje miejsce na wpisanie rozkazu. Po wpisaniu naciskamy klawisz RETURN, po czym na prawo BUG/65 wpisuje odpowiedni fragment kodu, również w hex i czeka na wprowadzenie następnego rozkazu. Oto jak przedstawiać się będzie na ekranie program, który napiszemy:

```
0600    LDA #27     A9 27
0602    STA 60C     8D 0C 06
0605    STA 60D     8D 0D 06
0608    INC 60D     EE 0D 06
060B    RTS         06
060C
```

W tym samym momencie naciskamy RETURN i wprowadzenie programu jest zakończone. Wyświetlmy go pisząc:

```
>D600 60D
0600 A9 27 8D 0C 06 8D 0D 06
0608 EE 0D 06 60 00 00
```

W tym małym programie posługując się poznanymi dotychczas rozkazami wprowadzamy do komórek o adresach 60C i 60D liczbę 27, a w drugiej z tych komórek zwiększamy zawartość o 1. Końcowy rozkaz RTS stosuje się, by po wykonaniu programu nastąpił niezakłócony powrót do BUG/65.

Z pomocą komendy "Y600 60D" spowodujemy, że na ekranie pojawi się niemal dokładnie to, co wpisaliśmy przedtem, a mianowicie rozszyfrowany przez BUG/65 zapis naszego programu w asemblerze. Zapamiętajmy tę użyteczną rolę komendy. W dwóch ostatnich liniach pojawią się rozkazy BRK. BRK - to rozkaz programowego przerwania, który ma kod 0. Dlatego debugger wpisuje ten rozkaz, chociaż nie jest naszym zamiarem wykonanie

go. Komórki 60C i 60D zarezerwowaliśmy na dane.

Spróbujmy teraz ten sam program zapisać z zastosowaniem prostych etykiet dostępnych w podręcznym asemblerze. Pomińmy prawą stronę ekranu, ponieważ asembler wpisuje tam dane wstępne, które będą korygowane.

```
0610      LDA #27
0612      STA L1
0615      STA L2
0618      INC L2
061B      RTS
061C      L1 BRK
061D      L2 BRK
061E
```

W dużych asemblerach z numeracją linii nie trzeba po etykiecie pisać BRK, tu natomiast jest to konieczne.

Teraz po naciśnięciu RETURN i wyświetleniu pamięci od 600 do 61D możemy przekonać się, że oba programy są analogiczne poza przesunięciem adresów o 10 w drugim z nich.

Czas na wykonanie naszego miniprogramu. Można to uczynić rozmaicie. Wybierzemy komendę "U", co oznacza "user run" - uruchomienie programu przez użytkownika. Piszemy:

```
>U600
>
```

Natychmiast w następnym wierszu pojawił się znak kursora programu BUG/65. Nasz program został wykonany. Sprawdźmy to:

```
>D600 60D
0600 A9 27 8D 0C 06 8D 0D 06
0608 EE 0D 06 60 27 28
```

W komórkach 60C i 60D zamiast zer pojawiły się wartości wprowadzone przez program.

Poznaliśmy na tym małym przykładzie najprostszą metodę pisania, wyświetlania i wykonywania programów oraz służące do tego komendy BUG/65 i ATARI Debuggera: Z, D, Y i U.

Jeszcze jedna użyteczna komenda będzie w ćwiczeniach często potrzebna. Jest nią S, substitute - zastąp , która pozwa-

la zmieniać zawartość komórek pamięci. Działa ona nieco inaczej niż poprzednie. Jeżeli np. chcemy zmienić zawartość komórki $601, piszemy: S601, po czym naciskamy s p a c j ę, a nie Return. Na ekranie pojawi się dotychczasowa zawartość komórki, w naszym przykładzie 27, i znak równości. W tym momencie wpisujemy nową wartość i znów naciskamy spację. Pojawia się wówczas zawartość następnej komórki ze znakiem równości, po którym znów możemy wprowadzać nowe dane. Return kończy działanie S.

Identycznie działa podręczny asembler w programie Atari Debugger. Podobne narzędzia dostępne są również na wszystkie pozostałe komputery oparte na mikroprocesorze 6502.

Ćwiczenia

x 1. Traktując akumulator jako zmienną A zapiszmy w Basicu przedstawiony wyżej przykładowy program.

x 2. Napiszmy w asemblerze w hex program odpowiadający na następującemu ciągowi instrukcji w Basicu (pamiętajmy o konwersji liczb z dec na hex) :

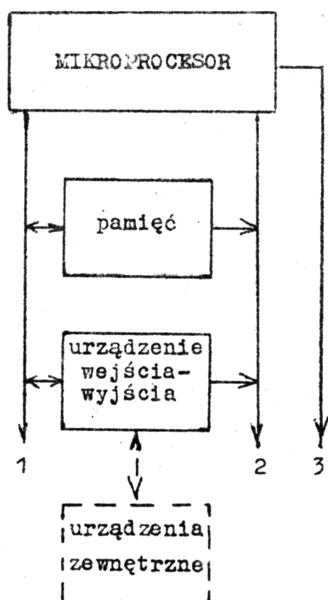
M=127:K=M:L=K+2

Rozdział 2

JAK DZIAŁA MIKROPROCESOR 6502?

2.1 Architektura mikrokomputera

W każdym mikrokomputerze można wyróżnić trzy podstawowe bloki funkcjonalne, które wraz ze schematycznym zaznaczeniem powiązań między nimi przedstawia rysunek 2.1. Są to: centralny procesor (CPU), pamięć (ang. memory) i urządzenie wejścia-wyjścia (ang. input-output unit - I/O).



Rys. 2.1 Architektura typowego mikrokomputera. Objaśnienia: A - szyna danych, B - szyna adresów, C - linie sterowania.

Centralny procesor pełni rolę głównego urządzenia przetwarzającego informacje i sterującego pracą mikrokomputera. W jego skład wchodzą dwa odrębne, lecz ściśle sprzężone urządzenia. Pierwsze z nich - to jednostka arytmetyczno-logiczna (ang. arithmetic-logical unit - ALU) wraz z wewnętrznymi rejestrami mikroprocesora. Najważniejszym z rejestrów jest akumulator, do którego z ALU przekazywana jest większość wyników obliczeń. Drugą ważną część składową mikroprocesora stanowi urządzenie sterowania (ang. control unit). Jedną z jego funkcji jest sterowanie wykonaniem kolejnych rozkazów programu.

CPU powiązany jest z resztą urządzeń mikrokomputera trzema rodzajami linii, zwanych szynami lub magistralami. Szyna danych (ang. data bus) umożliwia dwukierunkowy przesył danych między CPU a resztą urządzeń. 6502, podobnie jak inne mikroprocesory 8-bitowe, wyznacza jej rozmiar: można nią przesłać na raz 8 bitów czyli bajt. Szyna adresów (ang. address bus) określa swymi rozmiarami zasięg dostępu CPU do komórek pamięci. 6502 ma szynę 16-bitową, co oznacza możliwość dostępu do 65536 adresowanych komórek pamięci. Określamy to jako przestrzeń adresową CPU, która w danym wypadku wynosi 64 KB. Szyna adresów jest jednokierunkowa i służy CPU tylko do identyfikowania komórek.

Trzeci rodzaj połączeń nazwaliśmy liniami sterowania (ang. control lines), czasami nazywa się je również szyną. Jednakże linie sterowania ogarniają pajęczą siecią cały komputer i docierają do wszystkich jego części. Dlatego użyta nazwa wydaje się trafniejsza, a zaznaczanie punktów docelowych zbędne.

Drugim z podstawowych urządzeń komputera jest pamięć - układ wysoce skomplikowany, o którego istotnych dla programującego własnościach powiemy za chwilę.

Trzecie z wspomnianych urządzeń podstawowych zapewnia niezbędną w pracy komputera łączność ze światem zewnętrznym. W obrębie urządzenia wejścia-wyjścia wykonywane są dwie odrębne funkcje: wprowadzania i wyprowadzania informacji. W komputerach opartych na 6502 zastosowano formę tego urządzenia zwaną programowanym wejściem-wyjściem. Oznacza to, że "końców-

ki" urządzenia włączone są do przestrzeni adresowej analogicznie do zwykłych komórek pamięci. Z tego względu 6502 nie ma specjalnych rozkazów dla obsługi wejścia-wyjścia.

Między komputerami opartymi na 6502 występują różnice rozwiązań w dziedzinie obsługi wejścia-wyjścia, toteż kwestie te omówimy w rozdz. 9 dotyczącym niektórych właściwości Atari.

2.2 Pamięć

Zanim bliżej poznamy mikroprocesor 6502, przyjrzyjmy się pamięci - urządzeniu, bez którego nie zdołałby on zrealizować ani jednego programu. O pamięci komputera mowa była nieraz w rozdz. 1. Czas teraz zebrać podstawowe wiadomości i dorzucić następne. Pamięć służy do przechowywania informacji i umożliwia korzystanie z niej. Zorganizowana jest w komórki o rozmiarach bajtu oznaczone liczbowymi nazwami, które określamy jako ich adresy. Pamięć mikrokomputera można wyobrazić sobie jako regał z ponumerowanymi półeczkami, na których możemy umieszczać informacje. Na jednej półeczce można umieścić wartości od 0 do 255 dec. Komórki można wykorzystywać parami lub w większych całościach, co pozwala na operowanie dużymi liczbami.

W komputerze występują dwa podstawowe rodzaje pamięci wewnętrznej nazywane popularnymi skrótami RAM i ROM.

Pamięć zapisywalna, zwana też pamięcią o dostępie swobodnym (ang. random access memory - RAM) umożliwia zarówno odczyt (ang. read), jak i zapis (ang. write) informacji. Jest to zatem obszar pamięci, w którym użytkownik może rozmieszczać swoje programy i dane. Nikną one jednak po wyłączeniu komputera.

Drugi rodzaj pamięci nosi nazwę pamięci stałej (ang. read only memory - ROM). Z pomocą specjalnego procesu technologicznego programy i dane są w niej zapisywane na stałe, można je zatem jedynie odczytywać. Użytkownik nie może zmienić tego, co zostało zapisane na stałe. Informacje w ROM nie giną po wyłączeniu komputera i przy każdym jego użytkowaniu są dostępne w zawsze takiej samej postaci.

Istnieją również pośrednie rodzaje pamięci, PROM i EPROM, pozwalające użytkownikowi na zapisywanie z pomocą specjalnego

urzędzenia własnych programów lub danych - jeden raz na stałe lub wielokrotnie.

W ROM umieszcza się z reguły dane i programy w języku maszynowym stale użytkowane i wspierające mikroprocesor w wykonywaniu zadań obliczeniowych, koordynacji pracy wszystkich składników komputera oraz zapewnianiu jego współpracy z urządzeniami zewnętrznymi. Całość tych programów określa się zwykle jako system operacyjny (ang. operating system - OS) bądź jako monitor systemowy. Niezbędną częścią OS jest program zapewniający rozpoczęcie pracy komputera po jego włączeniu zwane inicjalizacją lub zimnym startem. Podczas zimnego startu system operacyjny sprawdza poprawność konfiguracji komputera, zeruje RAM, wprowadza do wielu komórek sterujących wartości początkowe, sprawdza, czy są przyłączone urządzenia zewnętrzne itd.

Ponadto w ROM znajdują się dane niezbędne do generowania znaków, a w większości mikrokomputerów także interpretator Basicu. ROM zawiera zatem setki większych lub mniejszych podprogramów, którymi również użytkownik może posłużyć się w swych programach. Szerzej omówimy te możliwości w następnych rozdziałach, zwłaszcza dziewiątym.

Rozmaicie rozwiązuje się sprawę programowej obsługi współpracy ze stacją dyskietek. W niektórych komputerach, np. w Commodore C64, program obsługi znajduje się w ROM, w innych, jak Atari, sprzętowo zapewnia się wykonanie najprostszych funkcji, natomiast resztę programu wgrywa się z dyskietki.

Różnice między RAM a ROM dotyczą charakteru dostępu do pamięci. Jednakże mikroprocesor traktuje oba obszary jako części jednolitej przestrzeni adresowej i zdolny jest wykonywać programy bez względu na ich położenie w pamięci.

Pamięć wewnętrzna komputera nie musi być ograniczona do rozmiarów jego przestrzeni adresowej, czyli w naszym przypadku do 64 KB. W wielu komputerach, np. w Atari 800XL i 130XE, Commodore C128 i innych, istnieją jej rezerwowe bloki zwane bankami pamięci, które można na zmianę włączać, a także wprowadzać do pewnych obszarów przestrzeni adresowej zamiast ROM - RAM. Sprawne przełączanie banków pamięci może stworzyć złudzenie, że mikroprocesor operuje na większej przestrzeni ad-

resowej. W rzeczywistości jej wielkość się nie zmieniła, a jedynie w pewnych jej obszarach jeden bank pamięci zamieniany jest na inny.

Ten typ dodatkowej pamięci przekraczającej wielkością pamięć operacyjną nosi nazwę pamięci wirtualnej. Jest ona dużym i coraz szerzej stosowanym udogodnieniem dla programisty. Dane zapisane w niej nie nikną po przyłączeniu innego banku i można do nich wielokrotnie sięgać, oczywiście, dopóki nie wyłączy się komputera lub nie nastąpi jego "zawieszenie się" na skutek ciężkiego błędu w programowaniu. Jednym z bardzo użytecznych sposobów wykorzystania dodatkowej pamięci jest zorganizowanie jej z pomocą odpowiedniego programu w swoistą dyskietkę wewnętrzną (ang. ramdisk).

2.3 Charakterystyka ogólna 6502

Sprawność mikroprocesora zwykło się oceniać na podstawie trzech cech: długości słowa danych, przestrzeni adresowej i szybkości wykonywania rozkazów.

Długość słowa danych w mikroprocesorze 6502 wynosi 8 bitów, a przestrzeń adresowa obejmuje 64 kilobajty.

Omówienia wymaga trzeci z wymienionych na wstępie parametrów, a mianowicie szybkość pracy. Stosuje się niekiedy jej porównywanie jedynie w oparciu o częstotliwość zegara wewnętrznego czyli generatora impulsów taktowych wyznaczających rytm pracy mikroprocesora, synchronizowany z pracą całego komputera. 6502, podobnie jak 6510 i 65C02, pracują w swej konfiguracji podstawowej z częstotliwością ok. 1 megahertza czyli ok. miliona cykli zegarowych na sekundę. Drugi popularny mikroprocesor 8-bitowy, Z80, realizowany jest w wersjach 2.5 i 4 MHz. Mogłoby się zdawać, że jest zatem dwa i pół raza lub czterokrotnie szybszy. Tak jednak nie jest.

Rzeczywistą miarą szybkości pracy mikrokomputera jest to, jak sprawnie wykonuje najprostsze operacje, z których składają się rozkazy. Operacje te noszą nazwę cykli rozkazowych. Otóż konstrukcja 6502 sprawia, że taką jednostkową operację wykonuje on w jednym takcie wewnętrznego zegara, czyli cykl rozkazowy jest równy cyklowi zegarowemu. Tymczasem

Z80 na jeden cykl rozkazowy zużywa kilka cykli zegarowych. Do zmniejszenia różnicy szybkości przyczyniają się również inne cechy 6502, które omówimy za chwilę. W efekcie tempo pracy obu procesorów jest podobne. Poszczególne rozkazy wykonywane są zależnie od stopnia ich złożoności w większej lub mniejszej liczbie cykli. Rozkazy 6502 wymagają 2 do 7 cykli zegarowych, a rozkazy Z80 - 4 do 21 cykli.

6502 zawdzięcza swą wysoką sprawność prostej, ascetycznej wprost organizacji wewnętrznej. Nie bez racji rozwiązania zastosowane w 6502 budzą uznanie informatyków, i rodzą oceny, że jest to najlepszy mikroprocesor w swej generacji.

Filozofia prostoty i przejrzystości znalazła w 6502 wyraz w zwięzłości jego listy rozkazów i ograniczeniu liczby rejestrów wewnętrznych przy jednoczesnej imponującej rozbudowie sposobów współpracy z pamięcią komputera wyrażającej się w zastosowaniu dużej liczby służących temu trybów adresowania. W pięciu z nich wykorzystuje się tzw. stronę zerową, czyli komórki o adresach 0-255. Tryby te pozwalają umieścić operand adresowy w jednym bajcie, skracają więc rozkazy i zwiększają szybkość wykonania. Ten wynalazek twórców 6502 sprawił, że ubogi w rejestry wewnętrzne mikroprocesor ma w istocie kilkadziesiąt dodatkowych rejestrów pracujących szybciej niż reszta pamięci. Na stronie zerowej można i należy umieszczać dane najczęściej wykorzystywane. Można na niej lokować ponadto 16-bitowe adresy. O szybkości pracy 6502 decyduje również duża liczba "szybkich" rozkazów jednobajtowych.

Być może, oceny te zabrzmią w tej chwili nieco abstrakcyjnie dla osób mniej wprowadzonych w temat. Warto do nich powrócić po dalszej lekturze.

Współcześnie, w związku z upowszechnianiem się mikroprocesorów 16-bitowych, a stopniowo także 32-bitowych, w sposobie projektowania mikroprocesorów zaznaczają się dwie odmienne tendencje. Jedna polega na zwiększeniu liczby i złożoności rozkazów przybliżających niejako język maszynowy do języków wysokiego poziomu. Druga, przeciwnie, zmierza do ograniczenia liczby rozkazów w celu zwiększenia szybkości ich wykonania. Buduje się komputery typu RISC (ang. reduced instruction set

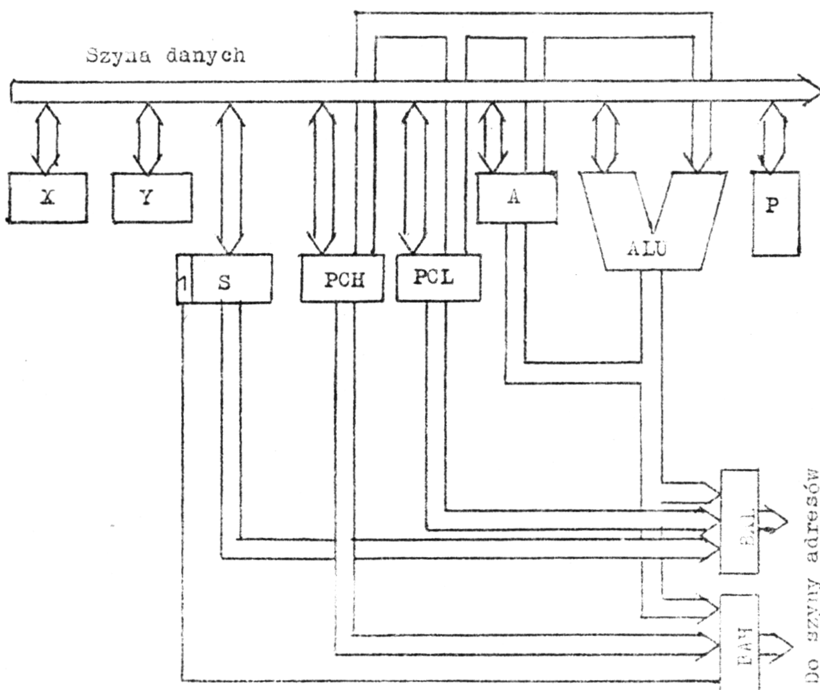
computer) o takich zredukowanych listach rozkazów. Drogę taką wybrał m.in. IBM tworząc najnowszą serię komputerów PS/2. Trudno w tej tendencji nie dostrzec rozwinięcia filozofii, która legła u podstaw 6502.

Skondensowany do 56 rozkazów język 6502 przy dużej elastyczności 13 trybów adresowania ułatwia i dyscyplinuje programowanie.

2.4 ALU i rejestry

Programujący w języku maszynowym nie musi w zasadzie wnikać w budowę mikroprocesora. Wystarczy mu znać narzędzia, do których ma dostęp z pomocą listy rozkazów. Coś nie coś warto jednak wiedzieć o podstawowych elementach mikroprocesora.

Omówimy pokrótce działanie jednostki arytmetyczno-logicznej ALU i rejestrów, posługując się rysunkiem 2.2.



Rys. 2.2 Budowa mikroprocesora 6502

ALU zgodnie z nazwą wykonuje operacje arytmetyczne i logiczne czyli przetwarzanie danych (ang. data processing). Można to urządzenie przedstawić, jak na rysunku, z pomocą litery V. Na dwa jej wierzchołki wpływają dane, a z dołu "wypada" wynik. Traktujmy ALU jako czarną skrzynkę, jej wnętrze nie będzie nas interesować. Nie jest ono dostępne dla programisty.

Wszystkie pozostałe części przedstawione na rysunku 2.2 - to rejestry i linie wewnętrznych połączeń mikroprocesora.

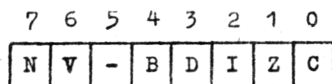
Wszystkie rejestry wewnętrzne 6502 są 8-bitowe, tylko wskaźnik stosu S ma dziewiąty bit. Także pod tym względem rejestry podobne są do pamięci. Jedna jest zasadnicza różnica: szybkości. Dostęp do komórek w pamięci w celu wpisania do niej lub odczytania z niej danych zajmuje mikroprocesorowi 300 nanosekund czyli 300 milardowych części sekundy. Do swoich rejestrów wewnętrznych 6502 dociera wielokrotnie szybciej. Rozkazy z udziałem danych w pamięci wykonywane są w ciągu co najmniej trzech cykli zegarowych na stronie zerowej i co najmniej 4 cykli poza nią, natomiast tam, gdzie chodzi o przeniesienie danych między rejestrami, o najprostsze, ograniczone do jednego bitu zmiany ich wartości, o przesunięcie proste lub cykliczne wszystkich bitów w rejestrze lub wstawienie do niego podanej bezpośrednio w rozkazie wartości (znak # przy liczbie w asemblerze), 6502 wykonuje polecenie w dwóch cyklach.

Z pozoru są to różnice nikłe, ułamki ułamków sekund. Przy większych obliczeniach przekonać się można, jak wiele znaczą i jak ważne jest umiejętne posługiwanie się rejestrami. Omówmy kolejno ich funkcje.

A - akumulator (ang. accumulator), bez wątpienia najbardziej pracowity z rejestrów 6502, jest specjalnym rejestrem ALU przyłączonym na stałe do jednego z jego wejść. Gdy ALU otrzymuje od linii sterowania zlecenie pracy, automatycznie szuka danych w akumulatorze, choć możliwe jest również jego ominięcie. W akumulatorze ALU pozostawia również przeważnie wynik obliczeń, stąd nazwa rejestru: akumuluje on wyniki.

P - rejestr znaczników zwany także rejestrem flagowym lub rejestrem stanu procesora (ang. processor status register) odgrywa specyficzną rolę: każdy bit w nim ma odrębne

znaczenie, toteż zwykło się go opisywać, nadając bitom literowe nazwy. Przedstawia to rysunek 2.3.



Rys. 2.3 Rejestr stanu procesora

Bity N, V, Z i C sygnalizują, że w ostatniej operacji, która została wykonana, powstał wynik o określonych cechach (1) lub że wynik cech takich nie ma (0). Pozostałe bity związane są ze sterowaniem pracą procesora. Znaczniki odgrywają podstawową rolę w sprawdzaniu warunków przy rozgałęzieniach czyli skokach warunkowych i tam ich rola będzie szczegółowo opisana. Tu podaję tylko angielskie nazwy, których skrótami są symbole literowe, oraz cechę sygnalizowaną, gdy znacznik przybiera wartość 1:

N - Negative. Wynik jest liczbą ujemną czyli jego bit b7=0.
V - Overflow. Wystąpił nadmiar, przeniesienie z b6 do b7 .
B - Break. Wykonano rozkaz BRK.
D - Decimal. Obliczenia wykonywa w binarnym kodzie dziesiętnym BCD.
I - Interrupt. Przerwania są zabronione przez programistę lub układy komputera.
Z - Zero. Wynikiem operacji jest 0.
C - Carry. Wystąpiło przeniesienie. Bit o wielu różnych zastosowaniach.

Bit b5 nie jest używany.

Umiejętne korzystanie ze znaczników, zwłaszcza N, Z i C, ma duże znaczenie. Cztery rozkazy służą do kasowania (zerowania) znaczników, a trzy do ich ustawiania (nadawania wartości 1). Również procesor wykorzystuje znaczniki w sterowaniu swą pracą,

Dwa kolejne 8-bitowe rejestry tworzą łącznie parę:

PC (ang. program counter) - to licznik programu zwany inaczej licznikiem rozkazów. Rejestr ten liczący łącznie 16 bitów zawiera adres, pod jakim znajduje się w pamięci instrukcja, która będzie wykonywana jako następna. Dwie części noszą

skrótowe nazwy PCL (PC low) - mniej znaczący bajt i PCH (PC high) - bardziej znaczący bajt licznika programu. Mikroprocesor po otrzymaniu adresu początku programu wstawia go do PC, a następnie automatycznie zwiększa go o długość kolejnych rozkazów. Zmienia też wartość PC po każdym rozkazie powodującym skok pod inny adres. Rolę PC omówimy szerzej w następnym punkcie rozdziału.

S - wskaźnik stosu (ang. stack pointer). Wskazuje następną wolną pozycję na stosie komputera. Rolę stosu omówimy w punkcie 2.6. Wskaźnik stosu jest rejestrem 9-bitowym z ustawionym na stałe na 1 najstarszym bitem. Oznacza to, że adresy, które można w nim przedstawić, mieszczą się w granicach od 256 do 511 dec czyli 100 - 1FF hex.

X i Y - dwa ostatnie rejestry przedstawione na rys. 2.1 noszą nazwę rejestrów indeksowych (ang. index register). Ich podstawowa rola polega istotnie na tym, że w sześciu trybach adresowania, po trzy z udziałem każdego z nich, umożliwiają sprawny dostęp do kolejnych adresów, ponieważ ich zawartość można zwiększać lub zmniejszać o 1.

Rejestry X i Y mogą być ponadto wykorzystane, podobnie jak akumulator, do czasowego przechowywania danych. Możliwy jest przesył danych w obu kierunkach między nimi a pamięcią, porównania i przekazanie danych do akumulatora. Pomysłowy programista z reguły stara się dać jak najwięcej "zajęcia" obu rejestrom. Sprzyja to tworzeniu krótszych i szybszych programów.

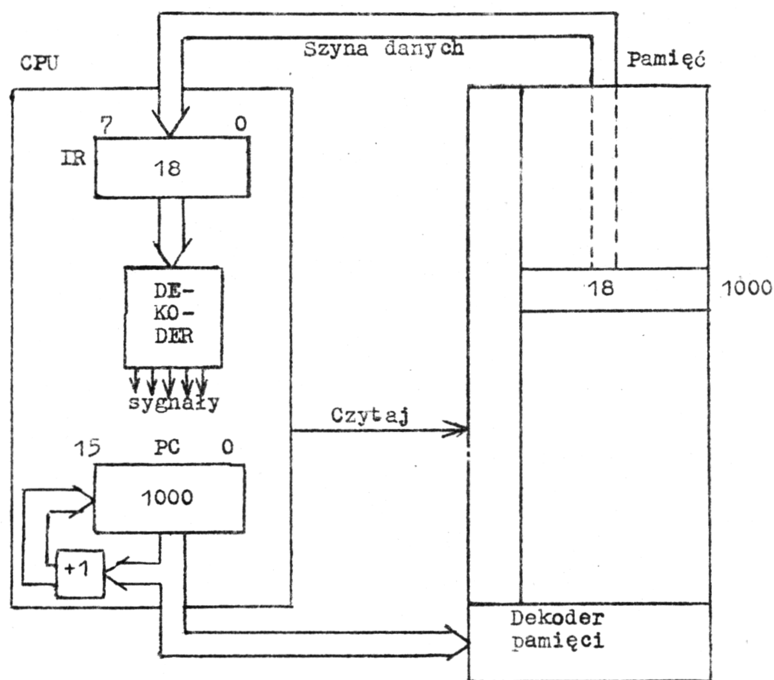
Rola rejestrów indeksowych będzie omówiona przy odpowiednich trybach adresowania. Pamiętać trzeba, że ich funkcje nie są w pełni symetryczne, każdy znajduje na ogół nieco odmienne zastosowania.

Na rys. 2.2 widoczny jest jeszcze jeden rejestr dwuczęściowy oznaczony symbolami BAL i BAH. Tego rodzaju rejestry służą do czasowego przechowywania danych i noszą nazwę b u f o r ó w (ang. buffers). W danym wypadku jest to bufor szyny adresów.

2.5 Cykl wykonania rozkazu

Sprawność mikroprocesora wynika w poważnej mierze z tego, że sterowanie wykonaniem rozkazów ma charakter zautomatyzowany, że w tempie mierzonym nanosekundami wykonywane są powtarzalnie trzy fazy wciąż tego samego cyklu:

- pobranie z pamięci następnego rozkazu do wykonania;
- zdekodowanie rozkazu;
- wykonanie rozkazu.



Rys. 2.4 Cykl wykonania rozkazu

Na rysunku 2.4, który poglądowo, choć w pewnym uproszczeniu przedstawia przebieg tego cyklu, pojawił się tylko jeden z poznanych poprzednio składników mikroprocesora, a mianowicie 16-bitowy licznik programu. Pozostałe należą do strefy dla programującego niedostępnej: do głęboko ukrytych przed nim ukła-

dów sterowania.

Prześledźmy kolejne fazy cyklu wykonania następnego rozkazu. Mówimy "następnego", a jest to w przypadku 6502 o tyle znaczące, że mikroprocesor ten, w przeciwieństwie do wielu innych, przystępuje do pobrania następnego rozkazu nie po wykonaniu poprzedniego, lecz już w czasie jego wykonywania. Jest to również jednym ze źródeł sprawności tego mikroprocesora.

Niech ów kolejny rozkaz, dla uproszczenia sprawy jednobajtowy o mnemoniku CLC i kodzie operacyjnym 18 hex, znajduje się pod adresem 1000. Mikroprocesor ma ten adres w liczniku programu. Podaje go na szynę adresów, skąd odczytuje go znajdujący się w pamięci dekoder adresów. Linią sterowania biegnie zarazem do pamięci polecenie "czytaj". Dekoder pamięci odnajduje komórkę o adresie 1000. Spowodowane zostaje odczytanie jej zawartości. W naszym przykładzie będzie to 18 hex. Liczba ta, oczywiście w postaci binarnej, bo tylko taką zna w swym obiegu wewnętrznym komputer, trafia na szynę danych, a mikroprocesor umieszcza ją w rejestrze rozkazów (ang. instruction register - IR). Zgodnie ze swą nazwą jest to rejestr, do którego trafia rozkaz przewidziany do wykonania.

Zakończyła się pierwsza faza.

Zaczyna się faza druga. Do pracy przystępuje układ dekodujący rozkaz. Zagląda do rejestru rozkazów i stwierdza, że jest tam 18 hex. Ustala, że oznacza to: CLC, clear carry, skasuj znacznik przeniesienia w rejestrze znaczników. Dekoder oblicza, że jest to rozkaz jednobajtowy i o 1 zwiększa zawartość licznika programu. Jest w nim teraz 1001. Koniec fazy drugiej.

Nowy rozkaz trafia teraz w gąszcz układów powodujących jego wykonanie. Czas trwania tej fazy zależy od charakteru rozkazu. Tym razem jest on stosunkowo prosty. Mimo to urządzenie sterowania musi wykonać parę kroków: znaleźć rejestr znaczników, wyzerować w nim najmłodszy bit - znacznik C. No, i wrócić z ważną wiadomością, że wszystko przebiegło pomyślnie. W przypadku bardziej złożonych rozkazów czynności jest znacznie więcej. Najbardziej rozbudowane w mikroprocesorach są z reguły układy sterowania wykonujące rozgałęzienia warunkowe.

Z różnic złożoności rozkazów wynikają znaczne różnice

w czasie wykonania. 6502 ma dużo prostych rozkazów wykonywanych, jak CLC, w obrębie rejestrów mikroprocesora. Te są dla niego najłatwiejsze.

Nasz przykładowy rozkaz został już wykonany. Tymczasem w rejestrze rozkazów znalazł się kolejny, spod adresu 1001. Automatyczny cykliczny proces wykonywania rozkazów potoczył się dalej. Zajrzyjmy do tablicy w aneksie, w której obok innych danych podaje się również liczbę cykli zegarowych wykonania rozkazów w rozmaitych trybach adresowania. Jest to miara czasowa pracy mikroprocesora, której mały fragment wymagał tak długiego opisu. A trwało to tym razem dwie milionowe częci sekundy. No, może nieco więcej. Może np. na mikrochwilę przerwały pracę procesora urządzenia wyprowadzające obraz na ekran. Wiele jest przyczyn tego, co nazywamy przerwaniami. Za każdym razem 6502 kończy wykonanie rozkazu, który ma na warsztacie, po czym głos oddaje innym urzędzeniom, by po zakończeniu przerwania wznowić pracę.

<u>Ćwiczenia</u>

x 1. Jakie rejestry ma 6502?

2.6 Koncepcja stron pamięci

Mikroprocesor 6502 traktuje pamięć komputera jako swego rodzaju książkę podzieloną na s t r o n y. Stronę pamięci stanowi fragment o długości 256 bajtów, a numerację stron zaczyna się od zera. Tak więc strona zerowa obejmuje komórki o adresach 0-255, pierwsza - o adresach 256-511, druga - o adresach 512-767 itd. Znacznie jaśniejsze stanie się to, gdy początkowe adresy kolejnych stron zapiszemy w hex. Będą to adresy:

0,100,200,300,400,500,600,...,1000,1100,...,2000 itd.

Jeżeli z dowolnego adresu zapisanego w hex odrzucimy dwie ostatnie cyfry, otrzymany numer strony. Tak więc MSB adresu oznacza numer strony, a LSB - położenie komórki na stronie. W przypadku części rozkazów JM przejście na inną stronę wydłuża czas wykonania o jeden cykl maszynowy.

<u>Ćwiczenia</u>

x 1. Jaki jest najwyższy możliwy numer strony pamięci?

2.7 Strona zerowa i stos

6502 wykorzystuje do swych celów trzy odcinki przestrzeni adresowej komputera: stronę zerową, stronę pierwszą oraz sześć najwyższych adresów w pamięci. Te ostatnie wiążą się z obsługą przerwań, co będzie omówione przy ich przedstawianiu.

Kluczowe znaczenie ma strona zerowa, ponieważ jej wykorzystanie jest absolutnie konieczne w pięciu ważnych trybach adresowania 6502, pozwala tworzyć kod szybszy i bardziej pamięciooszczędny. Zalety te sprawiają, że z komórek na stronie zerowej szeroko korzysta system operacyjny, a także interpretator Basicu. Użytkownikowi pozostaje niewiele miejsca i musi nim gospodarować racjonalnie, umieszczając na stronie zerowej najczęciej wykorzystywane dane, w tym adresy.

W każdym komputerze opartym na 6502 wykorzystanie strony zerowej rozwiązane jest odmiennie, toteż omówimy je na przykładzie Atari. Oto ogólny schemat strony zerowej Atari (adresy w hex):

00-7F - wykorzystywane przez system operacyjny
80-CA - wykorzystywane przez Basic
CB-D1 - dostępne dla użytkownika (203-209 dec)
D2-D3 - zarezerwowane dla Basicu
D4-FF - wykorzystywane przez pakiet operacji zmiennopozycyjnych OS.

Z mapy tej wynika, że przy korzystaniu z Basicu użytkownikowi pozostaje tylko 7 bajtów do dyspozycji przy tworzeniu podprogramów w JM. W rzeczywistości sytuacja jest korzystniejsza, ponieważ OS wykorzystuje szereg komórek tylko przy zimnym starcie komputera, np. bajty 0 i 1. Do wykorzystania są komórki 1D-1F, jednak po naciśnięciu RESET ulegają wyzerowaniu.

Po wyłączeniu Basicu, programujący może bezpiecznie wykorzystać adresy 80-D3 czyli 84 ($54) bajty. Ponieważ w programach w JM można zastąpić operacje zmiennopozycyjne znacznie szybszymi całkowitoliczbowymi, w każdym takim przypadku zwalniają się jeszcze bajty D4-FF, czyli dostępna staje się cała górna połowa strony zerowej. A to już jest niemało.

Na stronie pierwszej mieści się s t o s 6502. Przyjrzymy się bliżej tej ważnej strukturze.

Czym jest stos? Stanowi on fragment pamięci operacyjnej komputera przeznaczony do czasowego przechowywania danych, a nacechowany szczególną organizacją: dostępny jest mianowicie tylko obiekt, który był wprowadzony na stos jako ostatni. Mówimy o nim, że znajduje się na szczycie stosu.

Stos można porównać do talii kart położonej koszulkami do dołu na stole. Widzimy zawsze tylko jedną kartę znajdującą się na szczycie. Dopiero po jej zdjęciu widzimy następną, a potem kolejno dalsze. Jeżeli kilka kart położymy kolejno na talii, to zdejmować je potem będziemy w o d w r o t n e j kolejności. Inne porównanie: stos przypomina działaniem magazynek pistoletu, a także stos talerzy ułożonych jeden na drugim.

Taka zasada działania stosu określana jest po angielsku mianem LIFO - Last In First Out, czyli: ostatni wszedł, pierwszy wyszedł. Stosowa organizacja pamięci znana jest i wykorzystywana bardzo szeroko.Np. język programowania Forth wszystkie operacje na liczbach wykonuje za pośrednictwem stosu. Stos dla swoich potrzeb buduje również Basic.

Zasadnicza różnica między tamtymi stosami, a omawianym tutaj polega na tym, że gdy wcześniej wspomniane tworzone są z pomocą programów, to stos 6502 zarządzany jest sprzętowo przez mikroprocesor, jakkolwiek nie znajduje się w jego wnętrzu, lecz w pamięci.

8-bitowy wskaźnik stosu z dziewiątym bitem ustawionym na stałe ogranicza rozmiary stosu do 256 bajtów. Wymaga to umiejętnego posługiwania się stosem, lecz z drugiej strony przyspiesza operacje na nim. Można wykonać dwa zasadnicze typy takich operacji: zdjąć liczbę ze stosu (ang. pull) lub wstawić ją na stos (ang. push). W 6502 pierwszą czynność wykonuje m. in. PLA (pull accumulator from stack - zdejm liczbę ze stosu do akumulatora), a drugą m.in. PHA (push accumulator on stack - wstaw akumulator na stos).

6502 kontroluje stos z pomocą wskaźnika stosu S, który zawsze wskazuje pierwszą wolną pozycję na stosie. Początkową wartością S nie jest jednak 256, lecz 511. Należy bowiem pamiętać, że stos 6502, jak zresztą większość wykorzystywanych

stosów, skierowany jest szczytem k u d o ł o w i, czyli niejako odwrócony do góry nogami. Rozkaz PHA zmniejsza S o 1, a PLA zwiększa S o 1.

Z jednej strony stos wykorzystywany jest przez sam mikroprocesor i odbywa się to automatycznie bez udziału programującego, z drugiej zaś ten ostatni może również efektywnie posługiwać się stosem. 6502 wykorzystuje np. stos przy wszystkich skokach do podprogramów, by zapamiętać adres, pod który wrócić ma dalsza realizacja programu. Stos jest często użyteczny jako miejsce czasowego przechowywania danych. Z operacjami tego rodzaju zetkniemy się jeszcze wielokrotnie.

Należy przy tym pamiętać o dwóch ważnych rzeczach. Po pierwsze, po zakończeniu wszelkich operacji wykonywanych przez programistę na stosie musi być przywrócony stan, jaki miał on przed rozpoczęciem tych operacji. W przeciwnym wypadku komputer z reguły zawiesza się. Po drugie, czym innym jest stos, a czym innym wskaźnik stosu. Ten pierwszy jest w pamięci, drugi - w mikroprocesorze.

Ćwiczenia

x 1. Przedstawmy kolejne stany stosu i wskaźnika stosu po każdym rozkazie przy wykonywaniu następującego programu: LDA #27, PHA, PHA, LDA#30, PHA, PLA, PLA, PHA, PLA.

x 2 . Czy taki program działałby poprawnie?

Rozdział 3

KOMPUTEROWA ARYTMETYKA

3.1 Kody i liczby

Warunkiem skutecznego programowania w JM i asemblerze jest poznanie zasad, na jakich opiera się w komputerze przechowywanie informacji i jej przetwarzanie. Co więcej, nawet znanych ze szkoły czterech podstawowych działań arytmetycznych musimy się do pewnego stopnia "nauczyć" od nowa.

Fundamentalne znaczenie ma tu pojęcie k o d u, ponieważ w s z y s t k i e informacje w komputerze występują w postaci zakodowanej. Co to jest kod? Słownik podaje, że jest to "system umownych sygnałów, znaków liter, nazw itp. używany do przekazywania informacji". Kodem jest ludzka mowa, pismo, alfabet Morse'a, układy dziurek na taśmie perforowanej dalekopisu, fala radiowa niosąca informacje. Stosowanie kodów wymaga z reguły dwóch faz przekształcania informacji: jej przetłumaczenia na dany kod czyli zakodowania oraz czynności odwrotnej czyli zdekodowania.

W rozdziale 1 poznaliśmy sposoby przedstawiania liczb w układach pozycyjnych: dziesiętnym, którym najczęściej posługujemy się w życiu, dwójkowym czyli binarnym stanowiącym podstawową formę kodowania informacji w komputerze oraz szesnastkowym, najszerzej stosowanym przez programujących w asemblerze. Każdy z tych układów - to swoisty sposób kodowania liczb. Stosowanie rozmaitych reprezentacji liczb stawia na porządku dziennym problem ich przekształcania czyli konwersji z jednej reprezentacji na inną oraz sposobu wykonywania na nich działań arytmetycznych.

Tablica na rysunku 3.1 ukazuje charakterystyczne właściwości trzech układów liczbowych, o których była mowa. Lewa jej część obrazuje przekształcanie liczb binarnych w dziesiętne i szesnastkowe. Zwróćmy raz jeszcze uwagę, że każdą

N			2^N		
bin	hex	dec	bin	hex	dec
0000	0	0	1	1	1
0001	1	1	10	2	2
0010	2	2	100	4	4
0011	3	3	1000	8	8
0100	4	4	10000	10	16
0101	5	5	100000	20	32
0110	6	6	1000000	40	64
0111	7	7	10000000	80	128
1000	8	8	100000000	100	256
1001	9	9	1000000000	200	512
1010	A	10	10000000000	400	1024
1011	B	11	100000000000	800	2048
1100	C	12	1000000000000	1000	4096
1101	D	13	10000000000000	2000	8192
1110	E	14	100000000000000	4000	16384
1111	F	15	1000000000000000	8000	32768
10000	10	16	10000000000000000	10000	65536

Rys. 3.1 Liczby 0-16 i potęgi o takich wykładnikach w bin, hex i dec

4-bitową liczbę binarną można przedstawić z pomocą jednej cyfry hex, a zatem każdą 8-bitową dwiema cyframi hex.

Niemniej ciekawa jest prawa strona tablicy ukazująca właściwości poszczególnych układów liczbowych w przedstawieniu kolejnych potęg dwójki stanowiącej podstawę układu binarnego. 2 podniesione do potęgi 0 ma, jak w każdym układzie liczbowym, wartość 1. Potem wraz z każdym zwiększeniem potęgi o 1 na końcu liczby binarnej przybywa zero, ale pozostaje ona stale liczbą "okrągłą", złożoną z jedynki i samych zer. Można to wyrazić również inaczej: jeżeli jedynkę przesuniemy o jedną pozycję w lewo, to jej wartość podwoi się. Jeżeli zatem w liczbie binarnej o dowolnej długości wszystkie cyfry przesuniemy o jedną pozycję w lewo, będzie to równoznaczne z pomnożeniem tej liczby przez 2. I na odwrót, przesunięcie wszystkich bitów o jedną pozycję w prawo oznacza podzielenie liczby przez 2, przy czym reszta zostaje odrzucona.

Tę ważną właściwość liczb binarnych wykorzystują się wszystkie mikroprocesory, a wśród nich 6502. Ma on specjalne rozkazy powodujące przesunięcie wszystkich bitów w bajcie o

jedną pozycję w lewo lub w prawo. Na przykład, rozkaz o mnemoniku ASL (arithmetic shift left)- arytmetyczne przesunięcie bitów w lewo pozwala nieporównywalnie szybciej niż inne metody mnożyć liczbę przez 2, 4, 8 i dalsze potęgi dwóch. Jaki będzie wynik następujących operacji?

```
LDA #11
ASL A
ASL A
ASL A
```

Po trzykrotnym przesunięciu bitów akumulator zawierać będzie liczbę 11x2x2x2, czyli 88. Nawiasem mówiąc, poznajemy tu kolejny tryb adresowania zwany adresowaniem akumulatora (ang. accumulator addressing) stosowany jedynie dla czterech rozkazów przesuwania i tzw. obrotu bitów, czyli ich cyklicznego przesunięcia z udziałem bitu C w rejestrze znaczników. Rozkazy te można zastosować również w adresowaniu absolutnym i trzech innych trybach. Na rysunku 3.1 zwraca ponadto uwagę, że również w hex kolejne potęgi dwóch są liczbami okrągłymi o stałym repertuarze pierwszych cyfr: 1, 2, 4 i 8. Natomiast te same dane w układzie dziesiętnym są mało przejrzyste. Jest to jeszcze jeden argument za stosowaniem układu hex.

3.2 Przekształcanie liczb

W rozdz. 1 omówiono przekształcanie liczb binarnych na szesnastkowe i odwrotnie. Jest ono bardzo łatwe i wystarczy do niego kartka i ołówek. Nie ulega jednak wątpliwości, że układem najbardziej rozpowszechnionym jest dziesiętny i dlatego my albo za nas komputer musimy poznać skuteczne metody konwersji liczb binarnych na dziesiętne i na odwrót.

Poznana już metoda konwersji liczb binarnych na dziesiętne polega za sumowaniu liczb dziesiętnych odpowiadających w stosownej potędze jedynkom w liczbie binarnej. Im więcej jest cyfr w liczbie binarnej, tym wyższe wartości musimy sumować.

Jeszcze bardziej żmudna jest konwersja w odwrotnym kierunku: z dec na bin. Najprościej jest ją wykonać posługując się np. danymi z tablicy na rys. 3.1. Weźmy dla przykładu liczbę 35000. W tablicy odnajdujemy taką potęgę 2, którą mieści się

jeszcze w tej liczbie. Jest to 32768 dec czyli 2^{15}.

Zapisujemy ją jako liczbę bin oraz odejmujemy od naszej liczby wartość dec:

35000-32768=2232 1000000000000000

Najbliższe tej różnicy jest liczby 2048 czyli 2^{11}. Zapisujemy ją pod poprzednią i wykonujemy kolejne odejmowanie, po czym powtarzamy odejmowania aż do uzyskania wyniku 0. Pozostaje tylko zsumować częściowe wartości binarne. Oto całość tych czynności:

35000-32768=2232	2^{15}	1000000000000000
2232-2048=184	2^{11}	100000000000
184-128=56	2^{7}	10000000
56-32=24	2^{5}	100000
24-16=8	2^{4}	10000
8-8=0	2^{3}	+ 1000
		1000100010111000

Przekształciliśmy 35000 dec w 1000100010111000 bin. Konwersję można nieco uprościć wyznaczając na kratkowanym papierze pole liczby, tu z 16 kratek, i wpisując jedynki w odpowiednie kratki.

Podobny przebieg ma konwersja z dec na hex. Wykonajmy ją dla tej samej liczby 35000 dec. Posłużmy się poznaną tablicą, tylko zamiast wartości binarnych wpiszmy od razu częściowe wyniki w hex:

35000-32768=2232 dec 8000 hex
2232-2048=184 800

Dalej tablica okazuje się niewystarczająca. Trzeba wykonać dzielenie: 184:16=11 reszta 8. Nasza liczba - to 88B8. Do szybkiej konwersji można wykorzystać tablice w aneksie albo ... posłużyć się komputerem. BUG/65 i inne programy uruchamiające rozporządzają pomocnymi do tego komendami.

Do kompletu brakuje nam jeszcze jednej konwersji: z hex na dec. Weźmy uzyskaną przed chwilą liczbę 88B8. Trzeba pomnożyć jej kolejne cyfry przez odpowiadające im potęgi liczby 16 i zsumować iloczyny:

$8x16^3+8x16^2+11x16^1+8=8x4096+8x256+11x16+8=35000.$

3.3 Dodawanie i odejmowanie liczb szesnastkowych

Działania na liczbach są podobne we wszystkich pozycyjnych układach liczbowych, należy jednak pamiętać o istotnej różnicy: inne są podstawy układów. W układzie szesnastkowym przyzwyczaić się trzeba do istnienia dodatkowych cyfr A-F. Ich istnienie sprawia, że, nieco paradoksalnie na pierwszy rzut oka, 100 hex - to dwa razy więcej niż 80 hex. Rozpatrzmy krok po kroku dodawanie liczb hex. Wykonujemy je poczynając od skrajnej prawej cyfry.

```
 B485
+2C85
 E10A
```

a/ 5+5=A Nie 10, jesteśmy w innym układzie! Wpisujemy A.

b/ 8+8=10 Znów ta sama pułapka. Powstało przeniesienie. Wpisujemy 0.

c/ 4+C=1 z przeniesienia. C hex = 12 dec. Razem 17 dec czyli 11 hex. Powstało przeniesienie. Wpisujemy 1.

d/ B+2+1 z przeniesienia. B hex = 11 dec. Razem 14 dec czyli E hex. Wpisujemy ostatnią cyfrę.

W opisie pomocniczo posłużyliśmy się liczbami dec. Ideałem byłoby "myśleć heksadecymalnie", ale wymaga to wprawy.

Teraz odejmowanie w hex opisane także etapami:

```
 634F
-2E6C
 34E3
```

a/ F hex = 15 dec, C hex = 12 dec. F-C=3. Wpisujemy 3.

b/ 4 jest mniejsze niż 6. Pożyczamy 1 z wyższej cyfry. W hex oznacza to, że pożyczyliśmy 16, a nie 10. 4+16-6=14 dec czyli E hex. Wpisujemy E.

c/ W odjemnej po pożyczce zostało 2 i znów potrzebna jest pożyczka. E hex to 14 dec. 2+16-14=4. Wpisujemy 4.

d/ 5-2=3. Wpisujemy 3.

Dlaczego dość drobiazgowo zajęliśmy się prostymi działaniami w hex? Oswojenie się z tym układem jest konieczne chociażby dlatego, że wiele narzędzi programowania w asemblerze, np. programy uruchamiające, posługuje się niemal wyłącznie liczba-

mi szesnastkowymi. Ponadto znajomość hex - to krok wstępny do poznania działań w jeszcze jednym układzie liczbowym - binarnym czyli tym, w którym "myśli" komputer. Przyzwyczajmy się, że nasze tak swojskie liczby dziesiętne nie są jedynymi.

3.4 Dodawanie i odejmowanie liczb binarnych

Dodawanie dwóch c y f r binarnych daje następujące możliwe wyniki:

0+0=0

1+0=1

0+1=1

1+1=(1) 1

W ostatnim przypadku wystąpiło przeniesienie.

Omówmy kolejne kroki dodawania dwóch liczb binarnych:

```
 1010
+ 110
10000
```

a/ 0+0=0. Wpisujemy 0.

b/ 1+1=10. Powstało przeniesienie. Wpisujemy 0.

c/ 0 plus 1 plus 1 z przeniesienia = 10. Powstało przeniesienie. Wpisujemy 0.

d/ 1 plus 1 z przeniesienia daje 10. Dopisujemy z przodu 10.

Teraz odejmowanie:

```
 1100
- 110
  110
```

Podobnie jak w odejmowaniu liczb dziesiętnych potrzebne są nieraz pożyczki z wyższych cyfr.

a/ 0 minus 0 równa się 0. Wpisujemy ostatnią cyfrę.

b/ Pożyczamy 1. 10 minus 1 równa się 1. Wpisujemy 1.

c/ Znów potrzebna jest pożyczka. 10 minus 1 równa się 1.

Dwa dotychczas użyte przykłady dotyczyły niedużych liczb. Co się jednak stanie, gdy dodamy dwie liczby 8-bitowe? Oto przykład:

```
  10001000
 +10010001
1 00011001
```

Wiemy, że 6502 przetwarza liczby mieszczące się w jednym bajcie. Tu wynik wykroczył poza bajt, a to, co w nim pozostało, jest błędne. Jeżeli przeliczymy przedstawione dodawanie na wartości dziesiętne, to okaże się, że 136+145=25... Tu jednak dochodzi do głosu ważna właściwość mikroprocesora: w jego rejestrze P znajduje się znacznik C sygnalizujący przeniesienie. Dzięki temu fakt, że wynik wykroczył poza 8 bitów, nie zostanie przeoczony.

3.5 Dodawanie liczb 8 i 16-bitowych z pomocą ADC

W 6502 dodawanie wykonuje rozkaz o mnemoniku ADC, co stanowi skrót polecenia: add with carry - dodaj z przeniesieniem. Zastosujmy go:

```
LDA #27
ADC #31
STA B
```

Dwa pierwsze rozkazy powodują, że liczba 27 zostaje wprowadzona do akumulatora, a potem liczba 31 do niej dodana. Czynność tę wykonuje ALU. Gdzie zostawia wynik? Jak w większości przypadków, w a k u m u l a t o r z e. Zapamiętajmy tę ważną właściwość dodawania: zawsze jeden z jego składników musi znaleźć się w akumulatorze i zawsze w nim pojawia się wynik. Trzeci rozkaz sekwencji przeniósł wynik z akumulatora pod adres B. W Basicu odpowiadałaby temu instrukcja:

```
B=27+31
```

Wynik zdaje się być oczywisty: 58. A jednak ... A jednak tak być nie musi. Jedyny bowiem w 6502 rozkaz dodawania sumuje liczby z p r z e n i e s i e n i e m, czyli dodaje do nich zawartość bitu C w rejestrze stanu procesora P. Mogło się zdarzyć, że poprzednia czy nawet nieco wcześniejsza operacja spowodowała ustawienie tego bitu. Wtedy powstanie wynik 59, w oczywisty sposób błędny. 6502 rozporządza ważnym rozkazem zapobiegającym podobnym błędom: CLC czyli clear carry - skasuj znacznik przeniesienia.

I jeszcze jedna uwaga. 6502 rozporządza możliwością wykonywania dodawania i odejmowania liczb dziesiętnych w dostoso-

wanym do ich przedstawiania kodzie noszącym nazwę binarno-dziesiętnego (binary-decimal code - BCD). Przechodzenie z jednego kodu na inny wykonuje się ustawiając lub kasując specjalny znacznik w rejestrze P - znacznik D. Służy do tego para rozkazów SED, set decimal - ustaw tryb BCD i CLD, clear decimal - skasuj tryb BCD . W omawianym tu dodawaniu binarnym znacznik D musi być skasowany. Dlatego przed przejściem na arytmetykę binarną należy w zasadzie skasować znacznik D wprowadzając rozkaz CLD. Jeżeli wykonujemy więcej obliczeń, wystarczy tylko raz na początku zastosować CLD.

Działanie kodu BCD omówimy w punkcie 3.11.

Wracając do naszego programu dodawania, przed k a ż d y m dodawaniem binarnym musimy mieć pewność, że znacznik C jest skasowany. Dlatego poprawna postać naszego programu dodawania jest następująca:

```
DOD1  CLD
      LDA #27
      CLC
      ADC #31
      STA B
```

Przypomnijmy ponownie, że CLD wystarczy zastosować raz na początku obliczeń.

Dodawanie liczb 1-bajtowych można uznać za szczególny przypadek częstszego dodawania liczb 2-bajtowych. W tym wypadku znacznik C będzie nam potrzebny. Wykonajmy dodawanie dwóch liczb 16-bitowych przedstawionych w hex: A98D+297B. Zróbmy to najpierw na kartce wykorzystując poznaną już metodę:

```
 A98D
+297B
 D308
```

Zwraca uwagę, że między drugą a trzecią od prawej cyfrą, czyli na granicy bajtów nastąpiło przeniesienie. Oto poprawny program dodania tych liczb:

```
DOD2 CLD
     CLC       Skasowanie znacznika C
     LDA #8D   LSB pierwszej liczby w akumulatorze
     ADC #7B   Dodanie LSB drugiej liczby, wynik w akumulatorze
```

```
STA G      Zapamiętanie LSB wyniku pod adresem G
LDA #A9    MSB pierwszej liczby w akumulatorze
ADC #29    Dodanie MSB drugiej liczby, wynik w A
STA G+1    Zapamiętanie MSB wyniku pod adresem G+1
```

Teraz pod adresem G znajdzie się, zgodnie z ogólną metodą zapisu liczb dwubajtowych, mniej znaczący bajt sumy, a w następnej komórce - jej bardziej znaczący bajt. Taka kolejność bajtów nie jest konieczna, możemy zamienić je miejscami. Trzymajmy się tu jednak "zwyczajów" 6502.

Zastosowaliśmy cztery razy natychmiastowy tryb adresowania. Można jednak również założyć, że pierwsza liczba z przestawionymi bajtami znajduje się pod adresami E i E+1, a druga - pod adresam F i F+1. Zastosujemy wówczas tryb adresowania absolutnego:

```
DOD  CLD
     CLC
     LDA E
     ADC F
     STA G
     LDA E+1
     ADC F+1
     STA G+1
```

Zwraca uwagę, że w obu przypadkach tylko raz stosujemy CLC. Ponieważ rozkaz LDA nie zmienia wartości bitu C, możemy CLC umieścić zarówno przed LDA E, jak i bezpośrednio po nim. Co dzieje się natomiast, gdy dodawane są dwa bardziej znaczące bajty liczby? W tym momencie bit przeniesienia C ma wartość 1 i jest ona dodawana do sumy bardziej znaczących bajtów.

Ćwiczenia

1. Wykonajmy dwa pierwsze przykłady działań na odpowiednich liczbach dziesiętnych i sprawdźmy poprawność wyniku.

2. Dodajmy następujące pary liczb binarnych: 100010+10101, 1101+1011, 110110+1010.

x 3. Wykonajmy odejmowania: 110111-100100, 100110-1011, 100101-111.

x 4. Pierwsza liczba znajduje się pod K i K+1, druga pod

L i L+1. Wynik należy zapisać pod M i M+1. Napiszmy program, który wykonuje dodawanie tych liczb.

x 5. Czy zawsze wynik tego dodawania będzie poprawny?

3.6 Liczby ujemne i kod uzupełnienia do 2

Zanim przejdziemy do rozkazu odejmowania, rozważyć wypada dwie kwestie. Po pierwsze, wiadomo, że wszystkie cztery działania arytmetyczne można przedstawić z pomocą dodawania. Np. 3-5 jest równoznaczne z 3+(-5), a 8x4 z 8+8+8+8. Po drugie, wyłania się w związku z tym problem właściwego reprezentowania w komputerze liczb ujemnych. Pozornie wydaje się to proste: trzeba jakoś zaznaczyć znak + lub -, a pozostałą część liczby pozostawić niezmienioną, jak to czynimy w szkolnej arytmetyce. Okazało się, że kwestia właściwej reprezentacji liczb ze znakiem w komputerze nie jest bynajmniej tak oczywista i że "szkolna" metoda daje nienajlepsze efekty.

I oto w odniesieniu do mikrokomputerów zwyciężyła i jest dziś powszechnie wykorzystywana inna koncepcja: stosowania dla liczb ze znakiem specjalnego kodu noszącego nazwę uzupełnienia do dwóch. Powstały zatem dwa kody dla przedstawienia liczb liczb: jeden dla liczb bez znaku, a drugi - ze znakiem. Okazało się jednak, że ta pozornie bardziej skomplikowana metoda uprościła komputerową arytmetykę.

Czym jest ów kod uzupełnienia do dwóch zwany również krócej kodem uzupełnieniowym? Jego zasadę można poglądowo wyjaśnić na przykładzie licznika magnetofonu. Powiedzmy, że trzycyfrowy licznik wskazuje "000" i cofniemy go o jedną pozycję wstecz. Wskaże wówczas "999". Będzie to niejako wynik odjęcia 1 od 0 czyli -1.

Podobnie tworzy się liczby binarne w kodzie uzupełnienia do dwóch. Jeżeli z "00000000" cofniemy nasz "licznik binarny" o 1, wskaże "11111111". Jest to właśnie odpowiednik -1 w kodzie uzupełnieniowym. Reprezentacją -2 będzie binarne 11111110 itd.

Jak zmienić znak liczby, powiedzmy +57 czyli binarnie 00111001? Najpierw odejmujemy ją od 100000000, co jest równoznaczne z zamianą wszystkich zer na jedynki, a jedynek na zera. Nazywamy to i n w e r s j ą bitów. Powstająca liczba

nosi nazwę uzupełnienia do 1. Drugi krok - to dodanie 1. W ten sposób powstaje uzupełnienie do 2. Wykonajmy te czynności na naszej liczbie:

Wartość początkowa	00111001
Uzupełnienie do 1	11000110
Uzupełnienie do 2	11000111

Łatwo ustalić, że w kodzie uzupełnienia do 2 każdą liczbę o wartości - A reprezentuje 256-A.

Tablica na rysunku 3.2 przedstawia przykładowe liczby binarne w kodzie uzupełnieniowym oraz ich wartości bez znaku w dec i hex.

Liczba ze znakiem	Bin	Hex	Dec
-5	11111011	FB	251
-4	11111100	FC	252
-3	11111101	FD	253
-2	11111110	FE	254
-1	11111111	FF	255
0	00000000	00	0
1	00000001	01	1
2	00000010	02	2
3	00000011	03	3
4	00000100	04	4
5	00000101	05	5

Rys. 3.2 Liczby ze znakiem w kodzie uzupełnieniowym

Z tablicy wynika, że tę samą liczbę binarną mieszczącą się w bajcie można odczytać na dwa sposoby: jako liczbę bez znaku lub ze znakiem. W pierwszym przypadku zmieszczą się w bajcie liczby od 0 do 255, w drugim natomiast - z zakresu od -128 do +127. Różnica wartości powstaje przy tym dopiero powyżej 127, czyli od chwili, gdy najwyższy bit w bajcie przybierze wartość 1. Toteż w liczbach ze znakiem bit ten ma charakter bitu znaku: gdy ma wartość 0, liczba jest dodatnia, gdy 1 - ujemna. Po każdym działaniu arytmetycznym 6502 sprawdza wartość tego bitu w uzyskanym wyniku i identyczną przypisuje znacznikowi N w rejestrze P, znacznikowi wyniku ujemnego. Gdy posługujemy się arytmetyką ze znakiem, właśnie sprawdzenie

znacznika N pozwala ustalić, czy uzyskany wynik jest dodatni, czy ujemny.

Powstaje jednak istotny problem: liczba ze znakiem jest jakby krótsza o najwyższy bit i przy dodawaniu przeniesienie powstaje o jeden bit niżej. Dlatego właśnie w rejestrze znaczników istnieje jeszcze jeden bit sygnalizujący takie przeniesienie, które nosi nazwę n a d m i a r u. Z angielskiego słowa overflow wzięto dla nazwania tego znacznika drugą literę V, ponieważ O przypomina zero.

Znacznik V istotny jest tylko przy dodawaniu i odejmowaniu liczb ze znakiem. Spełnia wówczas ważną rolę w uzyskaniu poprawnego wyniku, sygnalizując przeniesienie z bitu b6 do b7.

Dochodzimy do niezwykle istotnej zalety kodu uzupełnieniowego. Oto dzięki niemu operacje na takich samych liczbach binarnych dają poprawne wyniki zarówno wtedy, gdy traktujemy je jako liczby bez znaku, jak i wtedy, gdy interpretujemy je jako liczby ze znakiem. Wymaga to jednak nieco odmiennego przeprowadzenia obliczeń.

3.7 Przeniesienie, nadmiar i pożyczka.

Rozpatrzmy przykłady dodawania ujawniające specyficzne właściwości kodu uzupełnieniowego. Wykonując dodawanie rozmaitych liczb notować będziemy stany znaczników C i V oraz wyniki dodawania liczb bez znaku i ze znakiem, a w tym ostatnim przypadku zbadamy również poprawność wyników.

Rozróżnić można trzy rodzaje sytuacji. Pierwsza polega na dodawaniu stosunkowo niedużych liczb, przy czym wynik nie powoduje ani przeniesienia, ani nadmiaru:

```
  00000111     5+7
+ 00001100    +12:
  --------    ----
  00010011    +19        C=0   V=0

  00000010      +2
+ 11111100      -4
  --------    ----
  11111110      -2       C=0   V=0
```

Jest to sytuacja, w której wynik jest poprawny identycznie jak przy dodawaniu.

Sytuacja druga polega na dodawaniu stosunkowo niedużych liczb, przy których powstaje przeniesienie do 9 bitu, ale nie

powstaje nadmiar.

```
  00000100    +4
+ 11111110    -2
1 00000010    +2        C=1     V=0

  11111110    -2
+ 11111010    -4
1 11111010    -6        C=1     V=0
```

Wynik jest poprawny pod warunkiem, że pominiemy przeniesienie.

Sytuacja trzecia polega na dodawaniu dwóch liczb dodatnich dających sumę większą niż 127, co powoduje nadmiar, a także dwóch liczb ujemnych dających sumę mniejszą niż 128, co również powoduje nadmiar.

```
  01111111    +127
+ 00000001      +1
  10000000    -128      C=0     V=1     Wynik niepoprawny

  10000001    -127
+ 11001110     -50
1 01001111     +79      C=1     V=1     Wynik niepoprawny
```

Ustalmy na tej podstawie zasady postępowania przy dodawaniu liczb ze znakiem:

- inaczej postępujemy z bitem przeniesienia zawsze go ignorując;
- bit nadmiaru V sygnalizuje konieczność skorygowania wyniku. Jak się to czyni, rozpatrzymy później przy omówieniu znacznika V oraz wykorzystujących go rozkazów rozgałęzień warunkowych BVC i BVS.

Pewna zawiłość działań na liczbach ze znakiem skłania wielu programistów do ich unikania, gdy jest to możliwe, i posługiwania się mniej "niebezpieczną" arytmetyką bez znaku. Np. Mansfield [2] ocenia wręcz rozkazy BVC, BVS i CLV jako nieprzydatne.

Ćwiczenia

x 1. Wykonajmy dodawanie poniższych par liczb binarnych ze znakiem, podając obok wartości składników i sum w dec oraz odpowiedzmy, które dodawania dają błędny wynik.

```
a/ 10100000      b/ 10000001      c/ 01111111
  +01100001        +11111100        +00000100
```

2. Czy może wystąpić nadmiar przy dodawaniu 8-bitowej liczby dodatniej i 8-bitowej ujemnej? Dlaczego?

3.8 Odejmowanie liczb 8 i 16-bitowych z pomocą SBC

Drugim obok ADC podstawowym rozkazem arytmetycznym 6502 jest rozkaz odejmowania SBC -(substract with borrow)- odejmij z pożyczką . W rzeczywistości rozkaz ten zamienia odejmowanie a-b na d o d a w a n i e a+(-b). Jak w każdym dodawaniu może tu zatem powstać przeniesienie i znacznik C będzie je nam sygnalizował. Jednakże w kodzie uzupełnieniowym - b będzie przedstawione jako 256-b. Warunek powstania przeniesienia można zatem zapisać jako

a+256-b > 256 czyli a-b> 0 czyli a>b.

Oto paradoks: przeniesienie powstanie wtedy, gdy nie ma pożyczki i na odwrót. Ta właściwość arytmetyki komputerowej sprawia, że aby uzyskać poprawny wynik odejmowania, należy przed nim znacznik C u s t a w i ć, a nie jak przed ADC skasować. Służy do tego rozkaz SEC set carry - ustaw znacznik przeniesienia . Pozy tym programy są analogiczne jak przy dodawaniu, przedstawione w punkcie 5.3. Oto ich wersje:

```
ODJ1  CLD           ODJ2  CLD           ODJ  CLD
      LDA #31             LDA #8D            SEC
      SEC                 SEC                LDA E
      SBC #27             SBC #7B            SBC F
      STA B               STA G              STA G
                          LDA #A9            LDA E+1
                          SBC #29            SBC F+1
                          STA G+1            STA G+1
```

W przypadku odejmowania liczb dwubajtowych lub dłuższych SEC należy, oczywiście, zastosować tylko w stosunku do pierwszego działania na najmniej znaczących bajtach. Potem, jak przy dodawaniu, nastąpi automatyczna korekta wyniku o przeniesienie.

Rozkazy ADC i SBC są w 6502 jedynymi rozkazami wykonującymi działania arytmetyczne na parach liczb. Nie ma stosowanych

w innych mikroprocesorach rozkazów dodawania i odejmowania bez przeniesienia, co wymaga opisanej korekty znacznika C. Nie rozporządza 6502 także rozkazami mnożenia i dzielenia, co jednak nie zaskakuje, bowiem ze względu na ich złożoność stosowane są w procesorach większych i ... droższych.

Do mnożenia i dzielenia potrzebne są odrębne podprogramy. Można jednak te działania uprościć, co pokażemy w punkcie 3.10.

Ćwiczenia

1. Napiszmy w asemblerze programy dla liczb 8-bitowych odpowiadające czynnościom opisanych w Basicu instrukcjami:

a/ Q=P-R x b/ Q=P+1-Q c/ K=L+1-M x d/ U=W+9-T

2. Jak wyrazić w Basicu poniższy program?

```
LDA N
SEC
SBC M+5
STA L
INC L
```

x 3. Co oznacza w tym programie M+5?

3.9 Zwiększenie i zmniejszenie o 1

Poznaliśmy wcześniej rozkaz INC. Zwiększa o 1 zawartość komórki pamięci. Należy do grupy rozkazów o podobnym działaniu, która obejmuje:

INC - zwiększenie o 1 zawartości komórki pamięci
INX - zwiększenie o 1 zawartości rejestru X
INY - zwiększenie o 1 zawartości rejestru Y
DEC - decrement memory by 1, zmniejszenie o 1 zawartości komórki pamięci
DEX - zmniejszenie o 1 zawartości rejestru X
DEY - zmniejszenie o 1 zawartości rejestru Y

Rozkazy te, wykonujące z pozoru drobną czynność, pozwalają dzięki trybom adresowania indeksowanego łatwo tworzyć konstrukcje o cyklicznym działaniu analogiczne do tych, jakim w Basicu służy:

```
FOR I=K TO L: ...  dalsze instrukcje  ...; NEXT I
```

Jest to znana praktycznie we wszystkich językach wysokiego poziomu pętla liczona oparta na sprawdzaniu wartości licznika pętli, którym w podanym przykładzie jest zmienna I. Gdy osiągnie ona wartość L, pętla zakończy działanie.

Drugim zastosowaniem indeksowania jest organizowanie dostępu do struktur danych zwanych w Basicu i wielu innych językach tablicami. W Basicu po zadeklarowaniu rozmiaru tablicy, np. z pomocą DIM A(10) otrzymujemy strukturę danych, w której możemy rozmieścić potrzebne wartości. Dostęp do nich uzyskujemy z pomocą indeksowania zmiennej A, np. A(0), A(1), A(2)... A(7) itd.

W celu odwzorowania pętli i tablic w asemblerze trzeba zastosować rozkazy, których jeszcze nie poznaliśmy, toteż odłóżmy sprawę na pewien czas. Widoczne jest natomiast, że rozkazy zmniejszenia i zwiększenia stanowią pewną namiastkę odejmowania i dodawania. Należy jednak pamiętać, że n i e wpływają one na stan znacznika przeniesienia. Gdy zatem wykonamy rozkazy

```
LDX#FE
INX
```

znacznik C nie sygnalizuje nam, że wynik przekroczył granicę bajtu. Natomiast możemy to sprawdzić z pomocą znacznika wyniku zerowego Z, który przybierze wartość 1, ponieważ w rejestrze X po dodaniu 1 do FF znajdzie się wartość 0.

Również znacznik wyniku ujemnego N reaguje na wyniki zastosowania każdego z opisanych sześciu rozkazów. Jeżeli np. w rejestrze Y będzie się znajdowała wartość 7F (127 dec) i zastosujemy INY, to znacznik N przybierze wartośc 1, bowiem w arytmetyce ze znakiem 128 ma postać binarną 10000000 i oznacza ... - 128.

Jak widać, znajomość zachowania się znaczników w rejestrze P nie jest całkiem prosta, a jest ona dla programującego w asemblerze niezbędna, ponieważ na niej opiera się poprawne wykonywanie wszelkich operacji arytmetycznych i logicznych, a także, co jest szczególnie istotne, sprawdzanie warunków i wykonywanie na ich podstawie odgałęzień warunkowych. By to przedstawić w oparciu o przykład, poznajmy wstępnie jeden z

ośmiu rozkazów odgałęzień czyli skoków warunkowych:

BNE - branch on non equal to 0 - wykonaj odgałęzienie, gdy wynik ostatniej operacji **nie jest równy zeru.**

Pary rozkazów LDX i STX oraz LDY i STY zapewniają rejestrom X i Y identyczne możliwości przesyłu, jak LDA i STA akumulatorowi. Rozpatrzmy następujący program (dane dec):

```
SUMA LDX #20     Inicjujemy rejestr X  wartością 20
     LDA #0      Inicjujemy akumulator wartością 0
     STA L       Przypisujemy zmiennej L wartość 0
     CLC         Kasujemy znacznik C przed dodawaniem
CYKL INC L       L=L+1
     ADC L       A=A+L, przez A oznaczamy akumulator
     DEX         Zmniejszamy X jako licznik pętli o 1
     BNE CYKL    Powtarzamy cykl, dopóki X nie równa się 0
     STA S       S=A
```

Jak działa ten program? Wykonuje on sumowanie wartości L z zawartością akumulatora, zwiększając L za każdym razem o 1. Czyni to 20 razy, gdy bowiem zawartość X zmniejszana po każdym cyklu o 1 osiągnie 0, rozkaz BNE przestanie powodować powrót do etykiety CYKL i ostatni rozkaz przeniesie zawartość akumulatora do S ($D2 czyli 210).

Innymi słowy program ten doda kolejne liczby 1+2+3+ ... +19+20 i w S umieścić sumę liczb całkowitych z tego przedziału. Skąd my to znamy? Ależ tak, to jest napisany w asemblerze przykładowy algorytm z punktu 1.6. Warto zwrócić uwagę na drobną, lecz bardzo istotną różnicę. Tam, w Basicu licznik pętli zwiększaliśmy za każdym razem o 1 i sprawdzaliśmy czy nie przekroczył 20. Tu licznik pętli, rejestr X, zmniejszaliśmy od 20 do 1, bowiem przy X=0 pętla nie została już wykonana. Dzięki temu nie trzeba było, jak w przypadku zwiększania X, sprawdzać, czy wskaźnik pętli przekroczył 20. W JM wymagałoby to dodatkowego, nie poznanego jeszcze rozkazu porównania CPX. Oszczędziliśmy jego 20-krotnego powtarzania czyli 60 dodatkowych cykli rozkazowych. Jest to ważna metoda optymalizacji pętli liczonych w JM.

Zwróćmy na koniec uwagę na istotny brak symetrii funkcji

między akumulatorem, a rejestrami X i Y. Akumulator uczestniczy w dodawaniach i odejmowaniach oraz gromadzi ich wyniki, natomiast nie można go zwiększyć ani zmniejszyć o 1 z pomocą INC lub DEC. W rejestrach X i Y możliwości te przedstawiają się odwrotnie, dostępne jest zmniejszenie i zwiększenie, nie można natomiast wprząc tych rejestrów do działań arytmetycznych.

Ćwiczenia

1. Napiszmy na nowo w Basicu program z punktu 1.6 uwzględniając nowy sposób liczenia pętli.

x 2. Jak zapiszemy w asemblerze poniższy program w Basicu?

```
10 S=0: FOR I=20 TO 1 STEP -1:S=S+I: NEXT I
```

3.10 Przesunięcie i obrót bitów

W punkcie 3.1 wspomnieliśmy o rozkazach przesunięcia i obrotu bitów oraz wstępnie poznaliśmy rozkaz ASL. Wymieńmy je wszystkie oraz opiszmy ich rolę:

ASL i LSR - logical shift right - arytmetyczne przesunięcie bitów w prawo różnią się tylko kierunkiem przesuwania bitów. Wspólne jest dla nich także to, że bit wypadający z liczby zostaje automatycznie przeniesiony do znacznika C. Ten pozornie drobny fakt przechowania traconego bitu ma, jak się przekonamy, duże znaczenie praktyczne. Warto zwrócić uwagę, że po ASL do C trafia najwyższy bit, a po LSR - najmniej znaczący.

Powiedzmy, że pod adresem G znajduje się binarna wartość 10000001 czyli 129 dec. Po ASL G liczba przekształci się następująco:

```
ASL 10000001
    00000010    C=1
```

Mówiliśmy wcześniej, że przesunięcie bitów w lewo oznacza pomnożenie liczby przez 2. Ale w komórce G tego nie widać, jest tam wartość 3. Możemy jednak wykonać:

```
LDA #0
ASL G
ADC #0
STA G+1
```

W ten sposób wykonując pozornie bezcelowe dodanie zera do zera w akumulatorze w rzeczywistości "łapiemy" utracony bit i teraz w dwóch bajtach G i G+1 nasza liczba jest zachowana z mniej znaczącym bajtem LSB jako pierwszym. Łatwo obliczyć, że ma ona wartość 1x256+2, czyli 258, czyli 2x129, o co chodziło.

Podobnie, a przecież inaczej, przebiega LSR G

```
LSR 10000001
    01000000   C=1
```

Skrajny prawy bit znalazł się w C. W komórce G znajduje się 64. Jest to wynik c a ł k o w i t o l i c z b o w e g o dzielenia G przez 2. W znaczniku C znalazła się reszta. Można ją przechować w komórce, powiedzmy, R (reszta), ale będzie miała zupełnie inne znaczenie.

Tą metodą można łatwo mnożyć i dzielić liczby przez kolejne potęgi 2. Na przykład sekwencja

```
LSR A
LSR A
LSR A
LSR A
```

umożliwia szybkie ustalenie wartości pierwszej cyfry hex, gdy wartość A chcemy przedstawić dwiema cyframi hex.

Można również w prosty sposób mnożyć liczby przez 3. Jeżeli chcemy wykonać to na liczbie w akumulatorze, trzeba posłużyć się pomocniczą zmienną TEMP

```
STA TEMP     Przechowuje w TEMP zawartość A
ASL A        Mnoży A przez 2
CLC
ADC TEMP     Ax2+A=Ax3
```

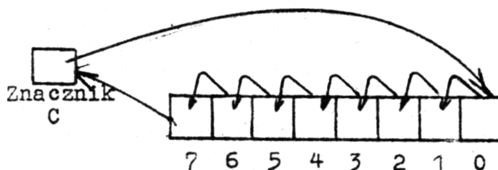
Nawiasem mówiąc do mnożenia liczby 1-bajtowej przez 256 wystarczą same rozkazy przesłań:

```
LDA G
STA G+1
LDA#0
STA G
```

Repertuar możliwości mnożenia i dzielenia znacznie rozszerzają dwa kolejne rozkazy noszące nazwę obrotu tzn.cyklicznego przesunięcia bitów. Są to:

ROL - rotate bits left, obróć bity w lewo
ROR - rotate bits right, obróć bity w prawo

Różnica w stosunku do poprzednich rozkazów polega na tym, że ROL i ROR nie tracą również poprzedniej wartości znacznika C wstawiając ją do zwalniającego się odpowiednio najniższego i najwyższego bitu. Tak więc w obrocie uczestniczy 9, a nie 8 bitów. Przedstawia to rysunek 3.3.



Rys. 3.3 Obrót bitów w lewo

Oto przykłady wykorzystania ROL i ROR w mnożeniu i dzieleniu przez 4 liczb dwubajtowych.

```
ASL G      Mnoży przez 2 mniej znaczący bajt
ROL G+1    Wstawia przeniesienie do najniższego bitu w G+1
ASL G      Ponownie mnoży mniej znaczący bajt przez 2 ...
ROL G+1    ... i ponownie wstawia przeniesienie.
```

Rozpatrzmy działanie tego programu na przykładzie liczby:

```
11000101    00100001
   G           G+1
```

Pierwszy, mniej znaczący bajt znajduje się pod G, bardziej znaczący w następnej komórce. Łączna wartość liczby wynosi $21C5. Kolejne rozkazy dotyczą za każdym razem jednego bajtu. Bit "wypadający" do znacznika C zapiszemy w nawiasie. Jego

wprowadzenie do G+1 zaznaczymy podkreśleniem. Oto kolejne kroki:

```
ASL G    (1)10001010   00100001   Najwyższy bit G wszedł do C
ROL G+1     10001010   01000011
```

Bit C został wykorzystany w G+1. Liczba ma teraz wartość $438A.

```
ASL G    (1)00010100   01000011   Znowu najwyższy bit wszedł do C
ROL G+1     00010100   10000111
```

Ta sama operacja została powtórzona. Liczba ma wartość $8714. Jest to wynik poprawny.

Dzielenie przez 4 liczby 2-bajtowej przeprowadzamy podobnie, tylko wobec innych bajtów, z przesunięciem i obrotem bitów w przeciwnym kierunku:

```
LSR G+1
ROR G
LSR G+1
ROR G
```

Z rozkazami ASL, LSR, ROL i ROR spotkamy się również w programach mnożenia i dzielenia dowolnych liczb.

Ćwiczenia

1. Analogiczną metodą, jak przedstawiona w poprzednim przykładzie zanalizujmy, jak odbywa się przesuw i obrót bitów w ostatnim przykładzie dzielenia przez 4. Sprawdźmy to na przykładzie $8716.

x 2. Czy wynik jest dokładny?

3.11 Kod binarno-dziesiętny (BCD)

Jak zbudowany jest wspomniany wcześniej kod binarno-dziesiętny (BCD)? Ta jeszcze jedna forma kodowania polega na tym, że w każdej połówce bajtu koduje się jedną cyfrę dziesiętną, a nie, jak w kodzie binarnym, jedną cyfrę szesnastkową. Tak więc najwyższą liczbą dziesiętną, którą można przedstawić w jednym bajcie, jest w kodzie BCD 99, a nie 255. Jak widać, kod jest znacznie bardziej rozrzutny pod względem wykorzystania miejsca, a wraz z tym obliczenia wykonuje się w nim powolniej.

Niech jego budowę unaoczni kilka przykładów liczb dziesiętnych i ich reprezentacji w BCD:

27	0010 0111
192	0000 0001 1001 0010
1987	0001 1001 1000 0111

Liczby dziesiętne w zapisie BCD przypominają zatem liczby w hex. Np. binarne $2718 wygląda tak samo, jak 2718 dziesiętne w BCD, ale próba dodania $AA i $BB w kodzie BCD będzie bezowocna, ponieważ kod ten nie zna takich liczb. Dodawanie i odejmowanie w kodzie BCD zewnętrznie przypominają już poznane. Należy podkreślić, że kod BCD zapewnia poprawne wykonanie tylko rozkazów ADC i SBC. Toteż tylko znacznik C zawiera sensowną informację. Natomiast znaczniki N, V i Z są bezużyteczne, a w związku z tym odgałęzienia warunkowe na ich podstawie, np. poznane BNE.

Z tych względów wielu programistów zaczyna program od CLD, by zabezpieczyć się przed niespodziankami. Znacznik D jest również skasowany po włączeniu komputera, ponieważ programy systemu operacyjnego, np. w Atari i Apple, pracują z reguły w kodzie binarnym.

Kod BCD znajduje zastosowanie zwłaszcza wtedy, gdy program posługuje się liczbami dziesiętnymi, np. przy przetwarzaniu informacji ekonomicznej czy finansowej. Co więcej, cała arytmetyka zmiennopozycyjna w Atari oparta jest na kodzie BCD. Jej zasady przedstawiamy przy omawianiu systemu operacyjnego tego komputera.

6502 umożliwia prostsze niż np. Z80 korzystanie z kodu BCD, ponieważ automatycznie dokonuje przeniesienia z niższego półbajtu do wyższego, gdy przy dodawaniu suma cyfr przekroczyła 9. W innych procesorach niezbędne jest przy tym często dokonywanie tzw. korekcji dziesiętnej. Przy przeniesieniach między bajtami trzeba wykorzystywać znacznik C.

Ćwiczenia

x 1. Do bajtu wpisano 9 dec. Wykonano czterokrotnie ASL. Jaką wartość dziesiętną ma liczba: a/ w kodzie BCD b/ w kodzie binarnym?

3.12 Kodowanie znaków

Z dotychczasowych rozważań wynika, że w ośmiu bitach składających się na bajtową komórkę pamięci można przedstawić:

- liczbę bez znaku
- liczbę ze znakiem
- część liczby 16-bitowej zajmującej dwa bajty
- kod rozkazu.

Komórka może również reprezentować z n a k ang. character czyli symbol graficzny o rozmaitym charakterze i zastosowaniu.

Komputery ogólnego przeznaczenia zapewniają możliwość wprowadzenia i wyprowadzania danych w postaci takich znaków. Znaki reprezentowane są w komputerze w postaci wielkości 1-bajtowych. Możliwych jest zatem 256 rozmaitych znaków.

Rozróżniamy trzy typy znaków:

- znaki cyfr od 0 do 9 oraz dużych i małych liter;
- znaki specjalne: przestankowe, działań arytmetycznych, "$" i in.
- znaki sterujące, które nie są drukowane, lecz powodują wykonanie rozmaitych czynności przy ich wyprowadzaniu, np. oczyszczenie ekranu, przeniesienie druku do nowego wiersza itd.

Znaki odgrywają zasadniczą rolę w przetwarzaniu tekstów.

Niestety, w sposobie kodowania znaków występują między komputerami opartymi na 6502 znaczne różnice. Podstawą w przestawianiu znaków cyfr i liter oraz znaków specjalnych jest na ogół kod ASCII określający wartości kodowe 128 znaków. Natomiast największe różnice między komputerami występują w wielkościach kodów znaków sterujących. W aneksie zamieszczone zostały wartości kodowe znaków Atari. Kod ten, nieco zmodyfikowany w stosunku do ASCII, nosi nazwę kodu ATASCII (Atari ASCII).

Mikrokomputery mają zawsze w swych systemach operacyjnych podprogramy powodujące wyprowadzenie znaku na ekran oraz czytanie znaku z klawiatury. W Atari pierwszy z nich znajduje się pod adresem OUTCHAR = $F2B0 (62128), a drugi pod adresem GETCHAR = $F24A (62026).

Wartość wyprowadzanego znaku musi znajdować się w akumulatorze. Przy czytaniu znaku z klawiatury podprogram czeka na jego wpisanie, po czym umieszcza kod również w akumulatorze. Np.

```
LDA $41
JSR $F2B0
```

spowoduje wyprowadzenie na ekran dużej litery A. W Atari istnieje również komórka o adresie $2FE (766). Wprowadzenie do niej wartości nie zerowej spowoduje, że znaki sterujące, z wyjątkiem RETURN (kod $9B czyli 155), będą wydrukowane, a nie wykonane.

Należy zawrócić uwagę na jeszcze jedną istotną dla programującego cechę. Oba podprogramy OS Atari wykorzystują rejestry X i Y mikroprocesora. Toteż jeżeli nasz program korzysta z tych rejestrów, należy znajdujące się w nich wartości przechować do chwili powrotu z podprogramu. Najłatwiej jest to uczynić na stosie, co będzie przedstawione w punkcie 4.1.

W Atari dzięki istnieniu drugiego procesora ANTIC możliwe jest również bezpośrednie wyprowadzanie znaków na ekran, o czym powiemy w rozdziale 9 omawiającym właściwości Atari. W jednym i drugim wypadku programista może wykorzystać opisane narzędzia do wprowadzania i wyprowadzania dużych tekstów.

3.13 Wnioski z treści rozdziału

Rozdział ten, zatytułowany "Komputerowa arytmetyka", zapoznał nas z wszystkimi rozkazami 6502 umożliwiającymi wykonywanie obliczeń arytmetycznych. Poznaliśmy dwa podstawowe rozkazy - dodawania i odejmowania - które temu służą oraz sposób ich stosowania wobec liczb 8 i 16-bitowych bez znaku. Zobaczyliśmy, jak można wykorzystać rozkazy zwiększenia i zmniejszenia o 1 oraz wykonywać z pomocą przesuwu i obrotu bitów najprostsze mnożenia i dzielenia. Zapoznaliśmy się z budową kodu uzupełnieniowego, w którym pracuje 6502, oraz z niektórymi metodami działań na liczbach ze znakiem.

Rozdział ten mógł zarazem przekonać uważnego Czytelnika, że jesteśmy jeszcze dość daleko od poznania wszystkich możliwoś-

ci obliczeniowych, jakimi rozporządza 6502. Nie poznaliśmy jeszcze w dostatecznym stopniu nawet metod, które trzeba zastosować w mnożeniu i dzieleniu. Rozkazy języka maszynowego realizują przeważnie wąskie i cząstkowe zadania. Sprawia to, że nieco bardziej złożone zadania wymagają opracowania algorytmów i programów.

Rozdział 4

PIĘĆDZIESIĄT SZEŚĆ ROZKAZÓW

4.1 Struktura listy rozkazów 6502

Poznaliśmy dotychczas około połowy rozkazów 6502, w tym większość najważniejszych i najszerzej stosowanych. Uporządkujmy tę wiedzę.

Rozkazy 6502 można podzielić według charakteru wykonywanych zadań na cztery następujące kategorie:

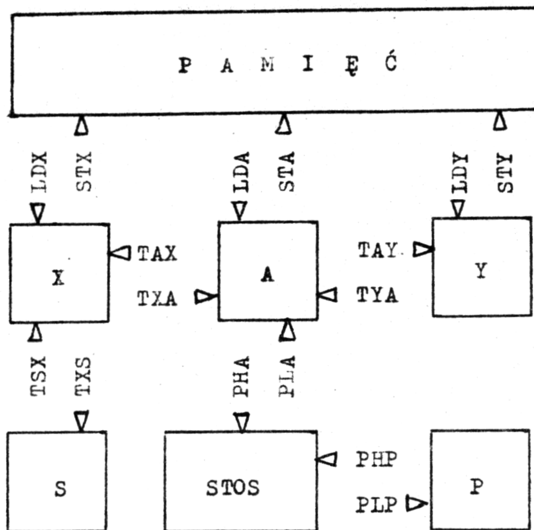
1. przesłania danych – 16 rozkazów
2. przetwarzania danych – 15
3. sprawdzeń, odgałęzień i skoków – 16
4. sterowania – 9 .

Te klasy rozkazów dostępne są we wszystkich komputerach ogólnego przeznaczenia. 6502 nie ma odrębnych rozkazów wejścia -wyjścia, ponieważ włącza jego zadania do ogólnej przestrzeni adresowej i wykonuje je tak jak na pamięci. Wejście obsługują rozkazy przesłania danych z pamięci do rejestrów CPU, a wyjście - rozkazy przesłania w przeciwnym kierunku, czyli zapisywania w pamięci.

4.2 Przesłania danych

Rysunek 4.2 przedstawia działanie wszystkich rozkazów przesłania dostępnych w 6502. Zapewniają one pełne możliwości przesyłu danych między pamięcią a rejestrami A, X i Y mikroprocesora, między CPU a stosem, a także między wewnętrznymi rejestrami 6502.

Przesłanie nigdy nie narusza zawartości komórki lub rejestru, skąd pobierane są dane.



Rys. 4.1 Rozkazy przesłania danych 6502

A. CPU - pamięć. Poznaliśmy już wszystkie rozkazy przesłania tej kategorii. Ładowanie rejestrów danymi z pamięci wykonują LDA, LDX i LDY, a zapis danych z rejestrów do pamięci - STA, STX i STY.

Rozkazy te, zwłaszcza LDA i STA, należą do najczęściej s stosowanych. Z ich pomocą można realizować, jak o tym była mowa, ważne funkcje analogiczne do tych, jakie w Basicu i innych językach wysokiego poziomu wykonują instrukcje przypisania wartości zmiennym bądź pobierania ich wartości do celów przetwarzania.

Nie ma **rozkazów powodujących** bezpośrednie przesłania pamięć-pamięć, wymaga to pośrednictwa rejestru, np.

```
LDA G          LDX L          LDY J
STA H          STX M          STY K
```

Z pomocą rejestrów wewnętrznych łatwo jest również dokonać wzajemnej wymiany zawartości między komórkami pamięci, co w Basicu odpowiadałoby przypisaniu zmiennej H wartości zmiennej

G, a G wartości H, np.: Z=G:H=G:G=Z Z jest tu zmienną pomocniczą .

```
LDX G
LDY H
STX H
STY G
```

B. Rejestr-rejestr. Istnieje komplet rozkazów dwustronnych przesłań między akumulatorem, a rejestrami X i Y. Łatwo zapamiętać ich mnemoniki, bowiem zaczynają się od "T", a następne dwie litery wskazują kierunek przesłania.

TAX - transfer accumulator to X, prześlij zawartość akumulatora do rejestru X. Dalsze rozkazy - to: TXA, TAY i TYA.

Istnieje również możliwość dwustronnego przesyłu wskaźnika stosu, jednak tylko z rejestrem X.

TSX - transfer stack pointer to X, prześlij wskaźnik stosu do rejestru X. Odwrotnie przesłanie wykonuje TXS. Pamiętajmy, że rozkazy te dotyczą wskaźnika stosu, a nie wartości na stosie.

Sześć omówionych rozkazów należy wraz z szeregiem innych do trybu adresowania implikowanego ang. implied zwanego też niejawnym lub wewnętrznym. Są to rozkazy jednobajtowe, a w samym kodzie operacji zawarta jest informacja, skąd i dokąd następuje przesłanie danych. Czas wykonania jest najkrótszy z możliwych - wynosi 2 cykle. Rozkazy te umożliwiają krótkotrwałe przechowywanie danych w nie używanym w danej chwili rejestrze. Jeżeli np. chcemy czasowo przechować zawartość potrzebnego w danej chwili akumulatora, a po wykonaniu przetwarzania mieć ją z powrotem w akumulatorze, umożliwi to sekwencja:

```
TAX       Przesyła A do czasowego przechowania w X
LDA G     A=G
CLC
ADC H     A=G+H
STA G     G=A=G+H
TXA       Przywraca poprzednią zawartość A
```

Akumulator nie zmienił zawartości, a po drodze wykonane zostało przypisanie: G=G+H.

C. Operacje na stosie. 6502 umożliwia wstawienie na stos zawartości dwóch rejestrów: akumulatora i rejestru znaczników P, a także odtworzenie A i P ze stosu. Komunikację akumulatora ze stosem zapewniają omówione już:

PLA - zdejmuje wartość ze szczytu stosu do akumulatora

PHA - wstawia na stos zawartość A.

Analogicznie odbywa się komunikacja P ze stosem:

PLP - pull processor status from stack, przenosi wartość ze stosu do rejestru znaczników

PHP - push processor status on stack, wstawia na stos zawartość rejestru znaczników.

Wszystkie cztery rozkazy odgrywają bardzo istotną rolę w rozwiązywaniu rozmaitych zadań programowych, zwłaszcza w przypadku skoków do podprogramów (JSR) oraz powrotów z nich (RTS). Jednakże te czynności wykonuje 6502 automatycznie. Natomiast programujący może wykorzystać zwłaszcza rozkazy PLA i PHA, do czasowego przechowywania danych na stosie. Niejednokrotnie, w chwili przejścia do podprogramu chcemy zapamiętać dotychczasową zawartość rejestrów. Właściwym miejscem dla takiego czasowego przechowywania danych jest stos. Ponieważ jednak komunikację ze stosem zapewnia tylko rejestr A, wymaga to zastosowania następującego ciągu rozkazów:

PHA	wstawia na stos zawartość A
TXA	przenosi do A zawartość X
PHA	wstawia na stos pośrednio zawartość X
TYA	przenosi do A zawartość Y
PHA	wstawia na stos pośrednio zawartość Y.

Po zakończeniu podprogramu, w celu odtworzenia poprzednich stanów rejestrów A, X i Y należy wykonać procedurę odwrotną. Pamiętać jednak trzeba o zdejmowaniu wartości ze stosu w odwrotnej kolejności. Tak więc wykona to ciąg rozkazów:

PLA	zdejmuje ze stosu zawartość Y
TAY	przenosi ją z A do Y
PLA	zdejmuje ze stosu zawartość X
TAX	przenosi ją z A do X
PLA	zdejmuje ze stosu zawartość akumulatora.

Efektem wykonanych operacji jest to, że w rejestrach A, X i Y znajdują się znów wartości, jakie były w nich w chwili skoku do podprogramu. Gdy, np.,rejestr X służył do wykonania pętli, tak jak w przykładzie programu przedstawionym w punkcie 3.9, wtrąciwszy dodatkowe czynności chcielibyśmy, by pętla była poprawnie kontynuowana. Ponieważ steruje nią rejestr , trzeba przechować jego wartość. Upraszcza to wysoce zautomatyzowany mechanizm stosu.

Stos odgrywa dużą rolę w obsłudze przerwań oraz w przekazywaniu danych czyli parametrów w Basicu do podprogramów w JM.

Ćwiczenia

x 1. Chcemy przechować czasowo na stosie zawartość komórek o adresach 1000 i 2000. Jak to wykonać?

2. Jak odtworzyć ze stosu zawartość, ewentualnie zmienioną komórek 1000 i 2000?

4.3 Przetwarzanie danych

Poznaliśmy już wszystkie rozkazy przetwarzania danych z wyjątkiem operatorów logicznych. Przypomnijmy pokrótce rozkazy dotychczas poznane.

6502 ma jedynie dwa rozkazy arytmetyczne: ADC i SBC - dodawania i odejmowania z przeniesieniem. Możliwość zastosowanie BCD podwaja w rzeczywistości liczbę rozkazów arytmetycznych.

Po trzy rozkazy umożliwiają zwiększenie o 1: INC, INX i INY, oraz zmniejszenie o 1: DEC, DEX i DEY.

Dostępny jest komplet rozkazów przesunięcia i obrotu bitów w lewo i w prawo: ASL, LSR, ROL i ROR.

4.4 Operacje logiczne

Lista rozkazów 6502 obejmuje trzy klasyczne operacje logiczne, których mnemoniki brzmią: AND, ORA i EOR. Rozkazy te mają szereg cech wspólnych:

- do ich wykonania potrzebne są zawsze dwa argumenty, czyli inaczej mówiąc są to operatory dyadyczne;
- jeden z argumentów znajduje się zawsze w akumulatorze, drugi może być wskazany bezpośrednio w trybie natych-

miastowym lub z pomocą adresu w operandzie rozkazu;
- wynik operacji pozostawiany jest w akumulatorze;
- operacje logiczne wykonywane są na pojedyńczych bitach liczb binarnych, a nie na całych tych liczbach.

Zwróćmy uwagę na tę ostatnią cechę. W językach wysokiego poziomu, np. w standardowych wersjach Basicu na Atari i Commodore, dostępne są jedynie operacje logiczne na parach liczb np. 123 AND 77 daje 1, 211 AND 0 daje 0. W związku z rosnącym znaczeniem operacji logicznych na bitach odpowiednie intrukcje pojawiają się coraz szerzej w językach programowania.

Działanie operatorów logicznych przedstawia tablica na rys. 4.1, zwana matrycą logiczną. Pokazuje ona wyniki operacji na argumentach, w danym wypadku bitach p i q.

p	q	p AND q	p ORA q	p EOR q
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Rys. 4.1 Matryca logiczna AND, ORA i EOR.

4.4.1 AND

Właściwością AND, czyli logicznego I koniunkcji jest to, że bit wyniku przybiera wartość 1 tylko wtedy, gdy oba bity mają wartość 1. Innymi słowy, jeżeli jednemu bitowi nadamy wartość 0, to wynik będzie zawsze równy 0. Nazywamy to m a s k o w a n i e m poszczególnych bitów czy całych fragmentów bajtu lub wielkości dwubajtowej, a stałą 8 lub 16-bitową, którą do tego wykorzystujemy - maską. Przyjrzyjmy się poniższej sekwencji rozkazów z argumentami napisanymi w hex. Obok ich binarne równoważniki.

```
LDA #C7     W akumulatorze jest 11000111 bin
AND #0F     Logiczne AND z      00001111 bin
STA B       Wartość z A jest w B 00000111 bin
```

Zastosowana maska OF sprawiła, że górna połowa bajtu została zamieniona w zera, natomiast dolna - powtórzona bez zmian. W praktyce oznacza to, że bajt ma obecnie wartość równą mniej znaczącej cyfrze hex. Sposób wyodrębniania pierwszej cyfry hex został omówiony w punkcie 3.10 (czterokrotne LSR). Stosując zatem przesunięcie bitów pierwszej cyfry i maskę $0F dla drugiej łatwo jest zbudować podprogram konwersji liczb dec na hex. Wrócimy do tego później.

Niejednokrotnie celowe jest stosowanie maski odsłaniającej tylko jeden bit. Np. Q AND 1 pozwala ustalić, czy liczba Q jest nieparzysta czy też parzysta. W odniesieniu do liczb ze znakiem Q AND $80 wyjaśnia znak liczby 1-bajtowej, a Q AND $8000 znak liczby dwubajtowej. W obu przypadkach wszystkie - bity liczby z wyjątkiem najwyższego zamieniane są w zera. Gdy zatem liczba jest dodatnia, wynikiem jest 0, a gdy ujemna - wartość niezerowa.

<u>Ćwiczenia</u>

1. Zapiszmy w postaci binarnej argumenty i wyniki AND na następujących parach liczb hex: a/ OF i 7 b/ FO i 9 c/ D3 i A5 d/ 74 i FF.

2. Co stanie się, gdy zastosujemy maskę AND $FF?

4.4.2 ORA

Mnemonik ORA zastosowano, by utrzymać konwencję mnemoników trzyliterowych, ale przypomina nam on również, że podobnie jak w pozostałych operacjach logicznych jeden z argumentów znajduje się w akumulatorze i tam umieszczany jest wynik.

Logiczne LUB czyli alternatywa, jaką wykonuje na bitach ORA, powoduje, że wszędzie tam, gdzie w którejkolwiej z dwóch liczb bit miał wartość 1, zachowa ją również w wyniku. Na przykład, ORA na wartościach binarnych

00010001 i
10001000

spowoduje, że w wyniku zachowane będą bity jedynkowe obu liczb. Powstanie zatem 10011001 bin.**Natomiast** ORA na 11010011 bin i 01011001 bin spowoduje, że w wyniku na każdej pozycji, na której bit w którejkolwiek liczbie miał wartość 1, zachowa ją,

tzn. w tym przykładzie wynik brzmieć będzie: 11011011 bin.

Ta właściwość rozkazu ORA powoduje, że maska wykorzystana przy nim nie zasłania bitów, natomiast pozwala dopisać nowe. Jeżeli użyjemy sekwencji liczby w hex :

```
LDA #A1
AND #OF
ORA #20
```

to po zasłonięciu z pomocą AND pierwszej cyfry hex ORA wpisze na to miejsce nową - tu zastąpi A przez 2.

ORA znajduje zastosowanie w przetwarzaniu znaków. Np. w Atari znaki o kodach powyżej $7F wyświetlane są na ekranie w postaci inwertowanej, tzn. znak przybiera barwę tła, a tło - barwę znaku, przy czym wartości odpowiednich znaków są o $80 wyższe. Oznacza to, że można uzyskać inwertowany znak wykonując na jego kodzie operację ORA z wartością $80, np.:

```
LDA #$41        Wprowadzono kod litery "A"
ORA #$80        Najwyższy bit zostaje ustawiony na 1
```

Ustawienie na 1 wartości najwyższego bitu stosowane jest często również w celu ułatwienia wyszukiwania nazw komend czy mnemoników rozkazów w ich tablicach w pamięci komputera. Zwykle zmienia się w tym celu kod ostatniego znaku w nazwie.

Są to niektóre z szerokich zastosowań ORA w programowaniu.

<u>Ćwiczenia</u>

1. Zapiszmy w postaci binarnej argumenty i wyniki ORA na następujących parach liczb zapisanych tu w hex: a/ 20 i 35 x b/ F6 i C5 c/ 55 i AA.

2. Spróbujmy wykonać w pamięci ORA na następujących parach liczb hex: a/ 35 i OF b/ 72 i FO c/ 04 i 81.

3. Jak zmieni się liczba, gdy wykonamy na niej: a/ ORA z 0 b/ ORA z FF?

4.4.3 EOR

Trzeci z rozkazów bitowego przetwarzania danych to EOR. Nie zawsze dostępny w Basicu pełni ważną rolę w JM. Przypomnijmy raz jeszcze jego matrycę logiczną:

p	q	p EOR q
0	0	0
0	1	1
1	0	1
1	1	0

Oznacza to, że EOR daje 1 tylko wtedy, gdy odpowiednie bity w dwóch liczbach binarnych mają r ó ż n e wartości. Ten rodzaj operacji logicznej nosi angielską nazwę "exclusive or", której skrótem jest mnemonik. W językach programowania, np. w Forth i C, stosuje się dla oznaczenia tej samej operacji nazwę XOR. W polskim nazewnictwie logicznym nazywa się ją ALBO, a także różnicą asymetryczną, nierównoważnością oraz zaprzeczeniem tożsamości.

EOR, podobnie jak AND i ORA, często wykorzystuje się ze stałą binarną jako maską. Np. z binarnym 00001111 czyli 0F hex EOR zmieni tylko prawą połowę bajtu.

Z pomocą EOR łatwo jest dokonać inwersji bitów w liczbie czyli przekształcić ją w jej uzupełnienie do jednego, jeżeli jako drugiego argumentu użyje się $FF. Oto przykład:

```
     01101001
EOR  11111111
daje 10010110.
```

Wystarczy do wyniku dodać 1, by uzyskać w kodzie uzupełnieniowym liczbę ze zmienionym znakiem, w naszym przykładzie zamiast 105 liczbę -105 dec, 10010111 .

Jeżeli wykonamy EOR liczby z nią samą, wynikiem będzie zero. Także stosując dwukrotnie wobec liczby taką samą maskę, otrzymujemy z powrotem tę liczbę. Np. sekwencja

```
LDA #07
EOR #2F
EOR #2F
```

sprawi, że w akumulatorze znowu znajdować się będzie 7. Po pierwszym EOR liczba przybierze binarną wartość 00101000, a po drugim 00000111 czyli 7.

Ćwiczenia

x 1. Wykonajmy EOR na parach liczb hex: 20 i 35, F6 i C7, 55 i AA, 72 i F0.

2. Jaki będzie wynik EOR liczby n z $00?

4.5 Porównania

Jedną z najważniejszych kontsrukcji programowania znanych praktycznie we wszystkich językach wysokiego poziomu jest konstrukcja o ogólnej postaci IF...THEN...ELSE lub o postaci skróconej IF...THEN. Można ją wyrazić zdaniem: "jeżeli warunek jest spełniony, wykonuj ciąg instrukcji S1, w przeciwnym wypadku wykonuj ciąg instrukcji S2". Widzimy, że podstawową przesłanką działania tej konstrukcji, jak i innych, pokrewnych, jest możliwość s p r a w d z a n i a, czy warunek jest spełniony, czy też nie.

W programowaniu sprawdzenie warunku polega najczęściej na p o r ó w n a n i u dwóch wielkości oraz stwierdzeniu, czy warunek jest spełniony, czy też nie, a używając pojęć logiki: czy wynikiem sprawdzenia jest prawda, czy też fałsz. Dążymy zazwyczaj do ustalenia, czy jedna wielkość jest równa drugiej, czy jest od niej większa lub mniejsza, czy jakaś zmienna przybiera wartość zerową czy też nie itd.

Porównania stosuje się również wtedy, gdy liczby reprezentują informację inną niż dotycząca liczb, np. są kodami znaków. Także znaki i słowa z nich złożone można przecież uporządkować odpowiednio do kolejności liter alfabetu w drodze porównania ich kodów.

Kwestie te wymagają wnikliwego rozpatrzenia w odniesieniu do JM i asemblera, ponieważ obok klasycznej formy porównania dwóch liczb wytworzono na tym najniższym szczeblu języków programowania szczególne formy sprawdzeń: ustalanie wartości

znaczników w rejestrze stanu procesora P.

Poznajmy sposób działania trzech rozkazów porównań dostępnych w 6502:

CMP - compare memory and accumulator, porównaj dane w pamięci lub podane bezpośrednio z akumulatorem;

CPX - compare memory and X register, porównaj dane z rejestrem indeksowym X;

CPY - compare memory with Y register, porównaj dane z rejestrem indeksowym Y.

Największe znaczenie ma rozkaz CMP dostępny aż w ośmiu trybach adresowania, czyli równie szeroko jak LDA, ADC, SBC, AND, OR, EOR. Do tej grupy rozkazów o najbardziej rozbudowanych formach adresowania należy również zaliczyć STA, który nie występuje z oczywistych powodów tylko w trybie natychmiastowym. Nie można przecież zapisać liczby ... w liczbie.

CMP ma podstawowe znaczenie w strukturach typu IF ...THEN oraz ON ... GOTO. Analogiczną rolę mogą pełnić CPX i CPY, jednak ich rola jest nieco ograniczona faktem, że rejestry X i Y są często stosowane do indeksowania pętli i innych celów, a CPX i CPY dostępne są w mniejszej niż CMP liczbie trybów adresowania.

Przy tych różnicach sam mechanizm porównania jest identyczny i warto go dokładnie zapamiętać. Polega on na tym, że wskazane dane są o d e j m o w a n e od zawartości rejestru A, X lub Y, lecz ani dane ani zawartość rejestru n i e są przy tym zmieniane. Natomiast stosownie do wyniku określane są znaczniki N, Z i C w rejestrze znaczników. Rozkazy porównań wykonują odejmowanie bez pożyczki, toteż zbędne jest SEC.

Dane mogą być podane w trybie natychmiastowym wprost w 1-bajtowym operandzie lub pobierane z pamięci na podstawie adresu zawartego w operandzie. Tak więc działanie rozkazów można przedstawić następująco, oznaczając dane jako D:

CMP: A-D   CPX: X-D   CPY: Y-D

Powtórzmy, że wynik odejmowania odzwierciedla się tylko w znacznikach rejestru P.

Powiedzmy, że w programie chcemy spowodować, by po naciśnięciu klawisza "A" następowało jego dalsze wykonanie. Rozwią-

zuje to sekwencja:

```
ZNAK JSR $F24A      Skok do podprogramu GETCHAR w OS Atari
     CMP#$41        Porównaj z kodem litery "A"
     BNE ZNAK       Jeżeli to nie "A", powrót do początku
                    pętli.
     ....
```

Gdy do programu wprowadzimy taką sekwencję, po dojściu do do niej wyprowadzanie danych na ekran zostanie zatrzymane do chwili naciśnięcia klawisza "A". Podobnie można opracować w Atari naciśnięcie innych klawiszy, w tym także badać zawartość komórki o adresie $D01F pozwalającej ustalić, czy naciśnięte są niektóre z klawiszy konsoli: OPTION, SELECT lub START.

Sprawdzenie warunku po rozkazach porównań z a w s z e w ostatecznym efekcie polega na sprawdzeniu stanu określonych znaczników sygnalizujących wynik porównania, jakie zostało ostatnio wykonane. Mechanizm ten będzie wyjaśniony w dwóch następnych punktach rozdziału.

Obok CMP, CPX i CPY mikroprocesor rozporządza dość oryginalnym, niezbyt często spotykanym w innych mikroprocesorach rozkazem porównania bitowego BIT (bit test). Podobnie jak rozkazy omówione poprzednio BIT nie zmienia zawartości akumulatora ani komórki pamięci, a wynik zapisuje tylko w znacznikach rejestru stanu procesora P. Wykonuje on logiczne AND na akumulatorze i komórce pamięci, po czym ustala wartość znacznika Z tak samo jak AND. Z przybiera wartość 1, gdy wynik jest 0, to znaczy, gdy w obu liczbach nie ma nigdzie na analogicznych pozycjach pary bitów 1. Na przykład, w zapisie binarnym pary batów:

```
    10101010        Wartość w akumulatorze
BIT 01010101        Wartość w pamięci
```

skasuje znacznik Z. Natomiast para:

```
    10101010        Wartość w akumulatorze
BIT 01001101        Wartość w pamięci
```

ustawi znacznik Z, ponieważ na pozycji b3 oba bity są jedynkowe. AND zapisałoby tu w akumulatorze 00001000 bin czyli 8. BIT nie zmieni zawartości akumulatora.

Ponadto BIT wykonuje dość nietypową czynność: kopiuje do dwóch najwyższych znaczników N i V wartości dwóch najwyższych bitów w komórce pamięci. K o p i u j e, a nie wstawia tam bitów wyniku. Tak więc w naszym ostatnim przykładzie efektem BIT byłoby: N=0, V=1, Z=0.

BIT jest jedynym rozkazem 6502, który ustawia znaczniki N i V nie na podstawie wyniku operacji. Właściwość tę można wykorzystać np. do szybkiego zbadania, czy liczba jest ujemna:

```
BIT Q
BMI UJEMNA
```

Gdy pod adresem Q w pamięci znajduje się liczba ujemna, to bez względu na wartość w akumulatorze znacznik N przybierze wartość 1 i nastąpi przeskok pod adres o etykiecie UJEMNA. Żaden inny rozkaz nie pozwoli wykonać tego tak łatwo.

Mimo niewątpliwych zalet użyteczność BIT jest ograniczona, ponieważ z jego pomocą nie można porównać wartości ze stałą w trybie natychmiastowym ani stosować adresowania indeksowanego.

4.6 Odgałęzienia czyli skoki warunkowe

W każdym języku programowania niezbędne są instrukcje umożliwiające zmianę sekwencyjnego porządku wykonywania programu w celu pominięcia, zależnie od warunku, pewnej jego części lub cyklicznego powtarzania wyodrębnionego fragmentu programu. W JM zapewniają to rozkazy odgałęzień (ang. branch) oraz skoków (ang. jump). Te pierwsze nazywane są również skokami warunkowymi, jednak użycie dwóch odrębnych nazw (odgałęzienia i skoki) jest uzasadnione ze względu na odmienność metody wykonywania owych przejść w inne miejsce w programie, zastosowanej w 6502.

Wspólną cechą obu typów rozkazów jest to, że powodują zmianę adresu zapisanego w liczniku programu PC mikroprocesora. Przypomnijmy, że PC wskazuje zawsze adres, pod którym w umieszczonym w pamięci programie znajduje się rozkaz, przewidziany do wykonania jako następny. PC jest automatycznie zwiększany o odpowiednią liczbę bajtów po rozpoczęciu wykona-

nia kolejnego rozkazu. Rozkazy odgałęzień i skoków zmieniają ten "naturalny" porządek.

Istnieje osiem rozkazów odgałęzień tworzących pary odpowiednio do wartości czterech znaczników. W każdej parze jeden rozkaz powoduje przejście, gdy znacznik przybierze wartość 1, a drugi w przeciwnym wypadku. Wartość znacznika określa tu w a r u n e k. Jeżeli nie jest on spełniony, przejście nie następuje i wykonywany jest następny rozkaz w programie. Oto rozkazy, obok podano znacznik i jego stan powodujący odgałęzienie:

BCC	C=0	i	BCS	C=1
BNE	Z=0	i	BEQ	Z=1
BPL	N=0	i	BMI	N=1
BVC	V=0	i	BVS	V=1

Wspólną cechą wszystkich rozkazów odgałęzień jest to, że reagując na odpowiednie znaczniki same n i e zmieniają wartości żadnego znacznika.

Jednolity dla wszystkich rozkazów odgałęzień jest sposób tworzenia nowego adresu w liczniku programu PC, odmienny niż w rozkazach skoków. Rozkazy odgałęzień w polu operandu zawierają tylko j e d e n bajt. Zawartą w nim wartość rozkaz traktuje jako liczbę z e z n a k i e m i dodaje ją do adresu n a s t ę p n e g o rozkazu. Operand ten nosi nazwę przesunięcia (ang. displacement lub offset). 8-bitowe przesunięcie ze znakiem oznacza możliwość cofnięcia się w programie o 128 bajtów lub przeskoku naprzód o 127 bajtów. Ponieważ jednak liczyć to trzeba od następnego rozkazu, to w chwili wpisywania rozkazu odgałęzienia możemy przewidzieć cofnięcie się o 126 lub skok naprzód o 129 bajtów.

Pisząc w asemblerze nie musimy wykonywać żmudnego obliczania 1-bajtowej wartości, jaka w kodzie maszynowym powinna być umieszczona jako operand. Wpisujemy po prostu adres lub etykietę, jak w przykładzie podanym przed chwilą. Czasem jednak chcemy wpisać wartość wprost w kodzie maszynowym i warto nabyć wprawy w jej obliczaniu, zwłaszcza dla skoków w tył. Oto przykład programu od adresu 600 hex:

```
0600   LDA #0        Załaduj 0 do A
0602   LDX #20       Załaduj 20 do X
0604   STA 4000,X    Wpisz pod adres 4000+X
0607   DEX           Zmniejsz X o 1
0608   BNE 604       Jeżeli X<>0, powtórz wpisywanie
```

Program ten wypełnia 20 hex kolejnych komórek pamięci wartością 0. Służy do tego zastosowany tu tryb adresowania absolutnego indeksowanego, przy którym zawartość X dodaje się do adresu w operandzie.

Odpowiedzmy na pytanie: jaka wartość znajdzie się w kodzie maszynowym jako operand przy BNE? Będzie to bez wątpienia wartość 1-bajtowa pod adresem 609 hex. Obliczmy ją. Od n a s t ę p n e g o rozkazu, czyli od adresu 60A, trzeba cofnąć się do 604 czyli o 6 bajtów. -6 w kodzie uzupełnieniowym to 256-6 czyli 250 dec, FA hex. Taką wartość wpisze asembler pod adres 609.

Ostatnią cechą wspólną rozkazów odgałęzień jest zmienny czas wykonania w cyklach zegarowych. Gdy odgałęzienie nie jest wykonywane, trwa 2 cykle, gdy jest wykonywane - o jeden cykl więcej, gdy na inną stronę pamięci - dodatkowo o cykl więcej. Dlatego pętli lepiej jest nie umieszczać na granicach stron, zwłaszcza gdy są wykonywane bardzo wiele razy.

W następnych podpunktach rozpatrzymy zachowanie się znaczników C, Z, N i V oraz ogólne zasady stosowania odpowiednich rozkazów odgałęzień.

Ćwiczenia

x 1. Napiszmy w asemblerze programy równoważne poniższym instrukcjom Basicu zastępując numery linii 20 etykietami L20.

a/ IF F+G=H THEN 20   b/ IF T+1<>U THEN 20

x 2. Jaki błąd tkwi w programie:

```
LDA 1000
BIT #1
BNE L20
```

4.6.1 Znacznik C, rozkazy BCC i BCS

Dużą rolę znacznika przeniesienia C poznaliśmy rozpatrując rozkazy arytmetyczne, obrotu i przesunięcia bitów oraz porównań. W operacjach arytmetycznych sygnalizuje on przeniesienie lub brak pożyczki. W przesunięciach i obrotach "przechwytuje" wypadający bit. W porównaniach zachowuje się tak, jak przy odejmowaniu.

Pamiętać trzeba, że C nie zmienia się pod wpływem żadnego rozkazu zwiększenia ani zmniejszenia, a także żadnego rozkazu przesłań poza PLP, który zastępując cały rejestr P wartością ze stosu może zmienić każdy ze znaczników. Na C działa również rozkaz powrotu z przerwania RTI, który omówimy dalej.

Z pomocą rozkazów CLC i SEC programujący może skasować lub ustawić C.

Tak więc znacznik C przybiera nową wartość odpowiadającą wynikom wykonania rozkazów: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

A co będzie się dziać, gdy w programie przez pewien czas nie zostanie zastosowany żaden z wymienionych rozkazów? Odpowiedź odnosi się do w s z y s t k i c h znaczników: w takim wypadku C, podobnie jak inne znaczniki, zachowywać będzie wartość, jaką otrzymał przy ostatniej dotyczącej go operacji. Dlatego właśnie przed dodawaniem trzeba go skasować, a przed odejmowaniem ustawić, by zapobiec błędowi, który może wywołać "utajona" wartość odziedziczona kiedy przez znacznik.

BCC, branch on carry clear - odgałęzienie, gdy C=0 i BCS, branch on carry set - odgałęzienie, gdy C=1 wykonują skoki odpowiednio do stanu znacznika przeniesienia.

Z rozkazami tymi zetkniemy się m.in. przy programach mnożenia i dzielenia. Pomocne są one również w korygowaniu możliwych błędów dodawania i odejmowania liczb 1-bajtowych. Przypuśćmy, że nasz program do liczby dwubajtowej umieszczonej pod adresami K i K+1 (mniej znaczący bajt jako pierwszy) dodaje liczbę jednobajtową L. Poprawność wyniku zapewni następujący ciąg rozkazów:

```
     LDA K       Młodszy bajt K w akumulatorze
     CLC         Skasowanie znacznika
     ADC L       Dodajemy L
     STA K       Wynik wpisujemy ponownie do K
     BCC DALEJ   Jeżeli C=0, wynik jest już poprawny
     INC K+1     Jeżeli C=1, starszy bajt zwiększamy o 1
DALEJ  następny rozkaz
```

BCC zapewni przeskok nad rozkazem INC K+1, gdy K+L nie przekroczy FF hex.

Analogicznie z pomocą symetrycznych rozkazów odejmujemy L od dwubajtowego K:

```
     LDA K
     SEC
     SBC L
     STA K
     BCS NAST
     DEC K+1
NAST  następny rozkaz lub koniec
```

Zwróćmy jednak uwagę na istotną różnicę. Gdy w drugim programie K=L, znacznik C pozostanie ustawiony.

BCC i BCS pełnią w asemblerze i JM analogiczne funkcje, jakie w Basicu można wyrazić z pomocą sprawdzenia warunku na nierówność liczb.

Sekwencję:

```
     LDA G       Dane spod G w A
     CMP H       Porównanie z danymi spod H
     BCC SKOCZ   Jeżeli dane > A, to idź pod adres SKOCZ
```

można traktować jako analogiczną do konstrukcji Basicu:

IF H > G THEN SKOCZ albo ogólniej: IF DANE > A THEN SKOCZ.

W przypadku sekwencji:

```
     LDA G
     CMP H
     BCS SKOCZ
```

występuje drobna, lecz istotna różnica. Odpowiada ona konstru-

kcji Basicu:

```
IF DANE≤A THEN SKOCZ
```

Łatwo bowiem zauważyć, że gdy H=G, także nastąpi skok.

Jeżeli zatem chcemy wykonać skok tylko wtedy, gdy H jest mniejsze od G, możemy operację zmodyfikować następująco:

```
LDX G
INX
CPX H
BCS SKOCZ
```

__Ćwiczenia__

x 1. Czy poprawna jest poniższa sekwencja?

```
     BCC SKOK
SKOK następny rozkaz
```

4.6.2 Znacznik Z, rozkazy BNE i BEQ

Z jest znacznikiem wyniku zerowego, toteż przybiera wartość jakby odwrotnie do wyniku: jest kasowany, gdy powstała wartość niezerowa, a ustawiany w przeciwnym wypadku.

Na znacznik Z wpływa znacznie więcej rozkazów niż na C, w tym wszystkie, które zmieniają wartość C. Dodatkowo znacznik wyniku zerowego sygnalizuje go przy większości przesłań (oprócz przesłań na stos, do pamięci i TXS), a także porównań, w tym bitowego BIT, zwiększeń i zmniejszeń.

Nie ma specjalnych rozkazów, które kasowałyby lub ustawiały znacznik Z. Łatwo jest to jednak osiągnąć. Np. LDA #0 spowoduje, że Z przybierze wartość 1.

Na stan znacznika wyniku zerowego wpływają następujące rozkazy: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA i TYA.

Na sprawdzaniu znacznika Z oparte są:

BNE - branch on not equal to 0, odgałęzienie, gdy wynik <>0.

BEQ - branch on equal to 0, odgałęzienie, gdy wynik=0.

W Basicu odpowiada to konstrukcjom:

```
IF DANE<>AKUMULATOR THEN SKOK     BNE
IF DANE=AKUMULATOR THEN SKOK      BEQ
```

Ponadto, jak już widzieliśmy, BNE pozwala łatwo skonstruowac pętlę liczoną analogiczną do konstrukcji Basicu:

```
FOR I=0 TO N:  ciąg instrukcji  : NEXT I
FOR I=N TO 0 STEP -1:  ciąg instrukcji  : NEXT I
```

Ogólną zasadą jest wykorzystanie w tym celu jednego z rejestrów indeksowych jako wskaźnika pętli. Ponieważ osiągnięcie przez ten wskaźnik granicy 0 można wprost kontrolować rozkazem BNE, natomiast osiągnięcie jakiejś wartości N wymaga porównywania z nią wskaźnika po każdym wykonaniu cyklu. zdecydowanie bardziej dogodny jest w JM odpowiednik drugiego rodzaju przedstawionej tu pętli. Wszędzie, gdzie można ją zastosować, trzeba to wykorzystać.

BNE i BEQ znajdują szerokie zastosowanie w programach. Oto drobny przykład. Mamy w pamięci poczynając od adresu T tablicę znaków o długości D i chcemy odnaleźć pierwsze wystąpienie w niej znaku cudzysłowu, którego kod wynosi 22 hex. W tym wypadku nie możemy badać tablicy od końca. Do kolejnych elementów tablicy uzyskamy dostęp z pomocą adresowania absolutnego indeksowanego.

```
      LDX #0       Początkowa wartość indeksu zmiennej T
NAST  LDA T,X      Element tablicy w akumulatorze
      CMP #22      Porównanie z kodem cudzysłowu
      BEQ ZNAL     Jeżeli znaleziono cudzysłów, przejście pod ZNAL
      INX          Zwiększenie indeksu zmiennej T
      CPX #D       Czy wskaźnik nie przekroczył długosci tablicy?
      BNE NAST     Jeżeli nie, badamy następny element
      .END         Komenda asemblera: koniec programu
ZNAL  STX WYNIK    Ostatni indeks zapisany w komórce WYNIK.
```

Program bada tablicę aż do adresu o 1 większego niż jej długość. Jeżeli w komórce o adresie WYNIK znajdziemy liczbę mniejszą lub równą D, będzie to wartość indeksu, pod którym w tablicy znajduje się kod cudzysłowu, w przeciwnym wypadku WYNIK=0 czyli cudzysłowu nie było. W programie użylismy dwóch

poznanych rozkazów odgałęzień. BEQ zapewniało przeskok do przodu w przypadku znalezienia poszukiwanego znaku, BNE - powtarzanie pętli do chwili, gdy cała tablica została zbadana. D nie może być większe niż 254.

Ćwiczenia

x 1. Czy można w przykładzie powyżej zastąpić BNE przez BCS? Czy coś się przez to zmieni?

2. Czy można tak przebudować ten program, by bez zmiany jego działania wyeliminować rozkaz CPX #D? (do sprawy wrócimy).

4.6.3 Znacznik N, rozkazy BPL i BMI

Znacznik wyniku ujemnego N zajmuje w rejestrze znaczników najwyższy bit, co zbieżne jest z faktem, że w liczbach ze znakiem właśnie w tym bicie liczby 8-bitowej lub w najwyższym bicie liczby 16-bitowej znajduje się informacja, czy liczba jest dodatnia, czy ujemna. W drugim wypadku bit znaku, jak zapewne pamiętamy, przybiera wartość 1. Tak więc znacznik N ma po wykonanej operacji taką samą wartość, jak bit znaku w wyniku. Niekiedy określa się go zresztą jako znacznik znaku (ang. sign flag) i oznacza literą S, co jednak może być mylące, ponieważ jako S oznacza się również wskaźnik stosu.

W większości wypadków znacznik N jest identyczny jak bit b7 akumulatora. Dzięki temu można sprawdzić ten bit w akumulatorze jednym rozkazem, gdy w przypadku pozostałych bitów nie jest to możliwe. Nie ma natomiast rozkazów pozwalających programiście zmienić znacznik N.

Zmianę znacznika N powodują w przypadku powstania wyniku ujemnego te same rozkazy, które mogą zmienić znacznik Z, gdy wynik jest zerowy. Mechanizm ustawiania i kasowania obu znaczników jest identyczny poza wspomnianym już wyjątkiem rozkazu BIT. Są to zatem rozkazy: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA i TYA.

Na sprawdzeniu znacznika N oparte są rozkazy:

BPL - branch on plus, odgałęzienie, gdy wynik dodatni

BMI - branch on minus, odgałęzienie, gdy wynik ujemny.

Ponieważ liczba 0 ma postać binarną 00000000, to zero traktowane jest jako liczba dodatnia i przy napotkaniu wyniku zerowego BPL powoduje przeskok.

Zdawałoby się oczywiste, że z pomocą BPL i BMI można po wykonaniu porównania sprawdzić warunek nierówności. Tak jednak nie jest. Przypuśćmy, że w akumulatorze jest 100 dec, a w pamięci 229 dec. Wynik odjęcia wykonanego przez CMP powinien brzmieć -129, ale wynik ten nie mieści się na 7 bitach i bez sięgnięcia po 9 bit przedstawić się go nie da. CMP porównuje jednak wartości 8-bitowe i wynik będzie błędny.

Dlatego też dla porównywania liczb b e z   z n a k u należy korzystać ze znacznika przeniesienia C, o czym była mowa. Porównywanie liczb z e   z n a k i e m - to obszerny temat, który omówimy w rozdziale 6.

4.6.4 Znacznik V, rozkazy BVC i BVS

Wspomnieliśmy już, że znacznik nadmiaru V sygnalizuje przeniesienie z bitu b6 do bitu znaku, które może spowodować błędny wynik przy dodawaniu i odejmowaniu liczb ze znakiem. W arytmetyce ze znakiem służy do korygowania wyników.

V można kasować z pomocą CLV. Nie ma rozkazu umożliwiającego ustawienie V. Ponadto na V oddziałuje BIT oraz, podobnie jak na wszystkie znaczniki PLP i RTI.

Znacznik V mogą zatem zmienić: ADC, BIT, CLV, PLP, RTI i SBC.

Na sprawdzaniu znacznika V oparte są:

BVC - branch on overflow clear, odgałęzienie, gdy V=0

BVS - branch on overflow set, odgałęzienie, gdy V=1.

Stosowanie tych rozkazów ogranicza się do arytmetyki ze znakiem. Poza nią nie są użyteczne.

4.7 Skoki

6502 rozporządza skokiem bezwarunkowym JMP (skrót słowa jump). Jest to rozkaz zawsze trzybajtowy i umożliwia dotarcie do każdego adresu.

Para rozkazów JSR i RTS zapewnia skok do podprogramu i powrót z niego.

Istnieje także rozkaz powrotu o specjalnym charakterze: RTI - return from interrupt, powrót z przerwania. Omówimy go w punkcie 9.7 poświęconym przerwaniom w Atari.

4.8 Rozkazy sterowania i pozostałe znaczniki

Ostatnia nieduża grupa rozkazów obejmuje w większości omówione rozkazy ustawienia i kasowania znaczników w rejestrze P, rozkaz przerwania wewnętrznego BRK, a także rozkaz NOP - no operation, który zgodnie ze swą nazwą nie wykonuje żadnej operacji i zwykle służy do tworzenia pauz.

Tak więc listę rozkazów 6502 uzupełniają: BRK, CLC, CLD, CLI, CLV, NOP, SEC, SED i SEI.

Istotna część procesów sterowania odbywa się w drodze ustawiania i kasowania znaczników w rejestrze stanu procesora P. Cztery z nich: N, V, Z i C, odzwierciedlają wyniki operacji wykonywanych z udziałem ALU i rejestrów. Trzy pozostałe: B, D, i I, mają szerszy zasięg działania. D pozwala ustawiać i kasować dodawanie i odejmowanie w kodzie BCD. Dwa pozostałe obejmują swym wpływem cały komputer i jego system przerwań. Programujący może skasować CLD i CLI znaczniki D i I lub je ustawić SED, SEI . Tylko dla znacznika B nie ma podobnych rozkazów. Ustawia go BRK, a skasować go można tylko na drodze programowej lub z pomocą klawisza RESET .

Uwaga. W aneksach Czytelnik znajdzie szczegółowe opisy działania i roli wszystkich rozkazów, ich kody operacji i inne użyteczne informacje.

Rozdział 5

TRZYNAŚCIE TRYBÓW ADRESOWANIA

5.1 Czym rozporządza programujący?

W poniższym zestawieniu wszystkich dostępnych na 6502 trybów adresowania podane zostały: polska i angielska nazwa trybu, długość rozkazu w bajtach oraz wzór zapisu przykładowego rozkazu z etykietą i operandem liczbowym w hex.

Etykiety oznaczają: #n - operand w trybie natychmiastowym, Q - adres dwubajtowy, Z - adres jednobajtowy wskazujący bajt lub słowo 2-bajtowe na stronie zerowej, L - przesunięcie ze znakiem w trybie względnym.

		Bajtów
1. akumulatora – accumulator		1
ASL A lub ASL		
2. implikowany – implied		1
TXA		
3. względny – relative		2
BNE L	BNE F9	
4. Natychmiastowy – immediate		2
LDA #n	LDA #41	
5. absolutny – absolute		3
LDA Q	LDA 4000	
6. strony zerowej – zero page, absolute		2
LDA Z	LDA D1	
7. absolutny indeksowany X – absolute indexed X		3
LDA Q,X	LDA 4000,X	
8. absolutny indeksowany Y – absolute indexed Y		3
LDA Q,Y	LDA 1500,Y	
9. strony zerowej indeksowany X – zero page indexed X		2
LDA Z,X	LDA D0,X	

10. strony zerowej indeksowany Y – zero page indexed Y 2
LDX Z,Y STX 25,Y
11. pośredni strony zerowej preindeksowany X – indirect zero page preindexed X 2
LDA (Z,X) LDA (15,X)
12. pośredni strony zerowej postindeksowany Y – indirect zero page postindexed Y 2
LDA Z ,Y LDA 15 ,Y
13. pośredni indirect – tylko JMP 3
JMP (Q) JMP 4000

5.2 Uwagi ogólne o trybach adresowania

Niemal wszystkie rozkazy JM do wykonania przewidzianych w nich czynności wymagają d a n y c h. Jedne rozkazy, jak ADC, SBC, CMP, wymagają dwóch argumentów, inne, jak rozkazy INC,ASL czy ROR - jednego, a jeszcze inne, przede wszystkim rozkazy przesłań, muszą otrzymać dane wyjaśniające skąd i dokąd należy przesłać dane. W rozkazach odgałęzień i skoków dane służą do ustalenia nowego adresu w samym programie.

Informacja o położeniu danych zawarta jest często w kodzie operacji. Np. TAX obok czynności przesłania jednoznacznie określa, skąd i dokąd mają być przesłane dane. Gdy potrzebne są dwa argumenty, położenie jednego z nich jest także z reguły określone w samym kodzie operacji, a miejscem tym jest przeważnie akumulator. W przesłaniach między pamięcią a rejestrami także określa się w samym kodzie operacji, skąd (np. STA) lub dokąd (np. LDY) mają być przesłane dane.

A gdzie jest drugi argument lub drugi adres? Takich informacji w samych kodach operacji nie ma w blisko połowie rozkazów 6502. Są one dostarczane w części rozkazu zwanej operandem. Sposób znajdowania owych brakujących danych nazywamy trybem a d r e s o w a n i a.

Jednobajtowe, nie mające operandów rozkazy należą do dwóch poznanych już trybów. Jeden - to stosowany tylko przy przesunięciach i obrotach bitów tryb akumulatora. Drugi - to tryb implikowany, w którym, jak w przypadku TAX, wszystkie informacje zawarte są w kodzie operacji.

W większości pozostałych trybów adresowania dostrzec można ważną cechę: istnienie trzech poziomów p o ś r e d n i o ś c i w odnajdywaniu danych. Owa pośredniość stanowi jedną z najważniejszych koncepcji programowania i została w poważnej mierze zrealizowana w 6502.

Pierwszy poziom stanowi tryb natychmiastowy, w którym dane podane są w samym operandzie, np. LDA=$D3.

Na następnym, drugim poziomie pośredniości dane nie są podane wprost, lecz wskazuje się ich a d r e s, np. LDA 1000.

I wreszcie trzecie i najwyższe piętro pośredniości reprezentuje jeszcze nie poznany tryb, a mianowicie:

LDA ($D3),Y

W trybie tym pod adresem $D3 na stronie zerowej, w tej i następnej komórce wskazuje się adres, pod którym, po dodaniu do niego zawartości Y, znajdują się potrzebne dane. Przyjmując dla uproszczenia, że Y=0, mamy tu sytuację, gdy miejsce danych wskazuje a d r e s a d r e s u.

Po co te zawiłości, czy sami sobie nie komplikujemy bez potrzeby życia? - zapyta ktoś. Praktyka przekonuje, że ów wysoki stopień pośredniości zapewnia programiście znaczne dodatkowe możliwości i w dużym stopniu ułatwia manipulowanie danymi.

5.3 Poznane tryby adresowania

Dokonajmy przeglądu wcześniej poznanych trybów adresowania, wskazując dostępne w nich rozkazy.

1. Akumulatora. Dostępne są w tym trybie rozkazy ASL, LSR, ROL i ROR. Są wówczas 1-bajtowe i wymagają 2 cykli. Rozkazy te można zastosować również w innych trybach.

2. Implikowany. Wszystkie rozkazy są 1-bajtowe i dotyczą operacji na rejestrach wewnętrznych 6502. Trwają dwa cykle, a gdy potrzebny jest także dostęp do pamięci - trzy (PHA, PHP), cztery (PLP, PLA), sześć (RTI, RTS), a nawet siedem (BRK) cykli. Wyłącznie na rejestrach wewnętrznych działają: CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA. Łącznie tryb ten obejmuje 25 rozkazów. Wy-

stępują one tylko w tym jednym trybie.

3. Względny. Obejmuje omówione w punkcie 4.6 dwubajtowe rozkazy odgałęzień. Ich wykonanie zabiera 3 cykle, gdy dochodzi do skoku, lub dwa w przeciwnym wypadku, a gdy skok wykonywany jest na inną stronę pamięci, dodać należy jeszcze jeden cykl.

Przypomnijmy rozkazy: BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS. Występują tylko w tym trybie.

4. Natychmiastowy. Wszystkie rozkazy są dwubajtowe, a wykonanie trwa dwa cykle. Operandem jest bajt danych. W trybie tym dostępne są rozkazy: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC. Łącznie 11 rozkazów.

5. Absolutny. Operandem jest pełny dwubajtowy adres komórki zawierającej dane lub adres skoku. Są to zatem rozkazy 3-bajtowe. Czas wykonania większości z nich wynosi 4 cykle, przesunięć i obrotów bitów oraz JSR - 6, JMP - 5 cykli.

Dostępne rozkazy: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY. Łącznie 23 rozkazy.

6. Strony zerowej (absolutny). Trybu tego nie omawialiśmy, jest on jednak prostym przeniesieniem na stronę zerową trybu absolutnego. Także w nim operand adresowy wskazuje bezpośrednio komórkę zawierającą dane. Ponieważ adresy na stronie zerowej mieszczą się w jednym bajcie, rozkazy w trybie adresowania strony zerowej zajmują dwa bajty. Skraca to czas ich wykonania o jeden cykl.

W trybie tym dostępne są te same rozkazy co w adresowaniu absolutnym z wyjątkiem JMP i JSR. W tych ostatnich adres na stronie zerowej musi być przedstawiany w postaci dwóch bajtów, z których starszy ma wartość 0.

5.4 Adresowanie absolutne indeksowane X i Y

Użyliśmy wprawdzie w jednym przykładzie tego trybu, celowe jest jednak jego szczegółowe omówienie. Należy do sześciu trybów 6502, w których stosuje się indeksowanie.

Sposób realizacji adresowania indeksowanego w 6502 jest odmienny niż w mikroprocesorach wyposażonych w bloki rejestrów

16-bitowych. 6502 ma tylko dwa 8-bitowe rejestry X i Y przydatne do tego celu.

Mimo istotnych różnic między poszczególnymi trybami indeksowanymi wspólną zasadą jest to, że w operandzie znajduje się informacja o adresie bazowym, do którego dodaje się zawartość jednego z rejestrów indeksowych uzyskując w ten sposób efektywny adres.

W adresowaniu absolutnym indeksowanym adres bazowy podany jest identycznie jak we wcześniej omówionym absolutnym czyli wprost w operandzie. Różnica polega więc tylko na tym, że do adresu dodaje się zawartość rejestru. Jeżeli, na przykład, rozkaz brzmi

```
LDA 1000,X
```

a w chwili jego wykonania w rejestrze X znajduje się wartość 83, to rozkaz spowoduje odnalezienie komórki o adresie 1083 (1000+83) i dane z niej zostaną załadowane do akumulatora. Jeżeli rozkaz brzmi

```
STA 1100,Y
```

a w chwili jego wykonania w rejestrze Y znajdują się 27, to zawartość akumulatora zapisana będzie w komórce o adresie 1127 (1000+27).

Adresowanie indeksowane zapewnia istotne ułatwienia w manipulowaniu danymi w pamięci. Rozpatrzmy to na przykładzie następującego programu kopiującego fragment pamięci w inne jej miejsce (dane w hex):

```
     LDY #E0       Górną granicę indeksu wprowadzamy do Y
CYKL LDA 2000,Y    Pobieramy dane spod adresu 2000+Y
     STA 3000,Y    Przenosimy je pod adres 3000+Y
     DEY           Zmniejszenie Y o 1
     BNE CYKL      Jeżeli Y<>0, powrót do początku spirali.
```

Efektem działania programu będzie to, że 240 bajtów poczynając od adresu 2000 hex zostanie skopiowanych na odcinek o 1000 hex wyższy. Tak proste wykonanie tej czynności nie byłoby możliwe, gdybyśmy nie rozporządzali indeksowaniem adresów. Ten sam indeks służy tu do równoległego posuwania się wstecz po dwóch ciągach komórek pamięci.

W przykładzie tym użyliśmy rejestru Y, ale z równym powodzeniem moglibyśmy wykorzystać tu rejestr X. Poznaliśmy zatem dwa tryby adresowania, które można stosować zamiennie z identycznym skutkiem.

Nie w każdym jednak wypadku taka zamiana jest możliwa, bowiem w korzystaniu z rejestrów X i Y nie ma pełnej symetrii.

Z rejestrem X można w adresowaniu absolutnym indeksowanym zastosować następujące rozkazy: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA.

Z rejestrem Y dostępne są: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA (ale niedostępne ASL, DEC, LSR, ROL, ROR).

Jak widać, tryb z wykorzystaniem X zapewnia większe możliwości. Czas wykonania poszczególnych rozkazów jest rozmaity, dla arytmetycznych i logicznych wynosi 4 cykle, przy czym przejście na inną stronę pamięci wymaga dodania 1 cyklu. Tak więc adresowanie absolutne indeksowane minimalnie tylko ustępuje szybkością absolutnemu, znacznie usprawnia natomiast dostęp do pamięci i przetwarzanie danych.

Ćwiczenia

x 1. Napiszmy w liczbach dziesiętnych w Basicu program, który wykona takie samo kopiowanie bloku pamięci, jak w przykładzie tego punktu.

5.5 Adresowanie strony zerowej indeksowane X i Y

Dwa kolejne tryby adresowania pozwalają zastosować opisane przed chwilą metody do danych na stronie zerowej. Nie znajdują one na tej stronie zbyt szerokiego zastosowania, bowiem trudno jest lokować tablice danych na tak niedużym obszarze, a właśnie przy takim zorganizowaniu danych najbardziej przydatne jest indeksowanie. Niemniej jednak są przykłady skutecznego wykorzystania tego trybu. Interpretator fig-FORTH na Atari właśnie na stronie zerowej lokuje stos główny i w jego obsłudze wykorzystuje indeksowanie.

Na stronie zerowej w pełni zachowało swą rolę, a nawet poszerzyło o rozkaz STY, indeksowanie z pomocą rejestru X. W tym trybie możliwe jest adresowanie rozkazów: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA,

STY.

Natomiast adresowanie strony zerowej indeksowane Y ograniczone zostało do dwóch rozkazów: LDX i STX. Możliwa jest zatem, na przykład, taka konstrukcja:

```
     LDY #3
CYKL LDX 0A,Y
     STX D0,Y
     DEY
     BNE CYKL
```

Program ten od adresu D0, umieści kopie zawartości komórek 0A-0D. W Atari w komórkach tych znajdują się wektory, czyli adresy, obsługi DOS, które nieraz warto przechować.

5.6 Trzeci stopień złożoności: adresowanie pośrednie

Przechodzimy do trzech ostatnich trybów adresowania 6502, które osobom mniej doświadczonym mogą się wydawać trudnymi, lecz otwierają przed programistą nowe możliwości. Chodzi o tryby adresowania p o ś r e d n i e g o, w tym pośredniego i n d e k s o w a n e g o.

W punkcie 5.2 wyjaśniliśmy, czym jest adresowanie pośrednie. Polega ono na tym, że dane odszukujemy na podstawie adresu ich adresu. Jedynym przedstawicielem czystego, nie indeksowanego adresowania pośredniego w 6502 jest rozkaz JMP. Pisze się go następująco:

```
JMP (000A)
```

W zapisie pojawiły się n a w i a s y. Jest to w asemblerach na 6502, 6510 i kompatybilnych powszechnie przyjęty sposób oznaczania trybu pośredniego. Czym różni się ten zapis od poniższego?

```
JMP 000A
```

Ten ostatni napisany jest w trybie absolutnym i powoduje, że skok w programie następuje do fragmentu zaczynającego się od adresu 0A czyli 10 dec. CPU powinien tu znaleźć kolejny rozkaz do wykonania. Powiedzmy od razu, że w Atari rozkazu nie znajdzie, ponieważ jest tam adres odczytywany, na przyk-

ład, przez Basic, gdy wywołujemy z niego DOS. Brak nawiasów nieuchronnie spowoduje załamanie się programu.

Natomiast JMP (000A) umieszczone w programie oznacza polecenie wykonania skoku pod a d r e s zapisany pod a d r e s e m 0A czyli w komórkach 0A (mniej znaczący bajt) i 0B (bardziej znaczący bajt). Oto istota trzeciego stopnia pośredniości, o którym mowa była w punkcie 5.2, a zapewne również trzeciego stopnia trudności w poznawaniu trybów adresowania 6502.

Szukając analogii w Basicu zastąpmy adres numerem linii. Nasz skok pośredni można wówczas przedstawić następująco:

```
100  A=10:GOTO A
```

Także tu adres skoku w postaci numeru linii podany jest pośrednio: w zmiennej.

Duże procesory mają nieraz bardzo rozbudowane tryby adresowania pośredniego. W 6502 nie ma tak szerokiej ich gamy. Podkreślmy jednak, że 6502 jest jednym z niewielu mikroprocesorów 8-bitowych, kt re rozporządzają takim sposobem adresowania.

Pośredniość adresowania może mieć jeszcze wyższe stopnie: odczytywanie adresu za pośrednictwem adresu adresu itd. Programujący w asemblerze zdolny jest zbudować takie struktury i efektywnie z nich korzystać. Przedtem jednak warto poznać możliwości tkwiące w mikroprocesorze, bo on mimo wszystko wykona zadania najszybciej.

6502 rozporządza następującymi możliwościami adresowania pośredniego:

- jest jeden, omówiony już, rozkaz JMP dostępny w trybie pośrednim z operandem dwubajtowym;
- wszystkie adresy dla dwóch pozostałych trybów adresowania pośredniego muszą znajdować się na stronie z e r o w e j;
- w trybach tych dopuszczalne jest jedynie adresowanie i n d e k s o w a n e z pomocą rejestrów X i Y
- sposób indeksowania z pomocą każdego z tych rejestrów jest i n n y. W asemblerze zaznacza to różnica zapisu: X w nawiasie, a Y poza nim .

5.7 Z rejestrem X - preindeksowanie

Adresowanie pośrednie preindeksowane X zapisuje się w asemblerze następująco:

```
LDA (Z,X)
```

Rozpatrzmy następujący przykład (dane w hex). Na stronie zerowej pod adresami CC-D1 umieszczone są trzy pary liczb:

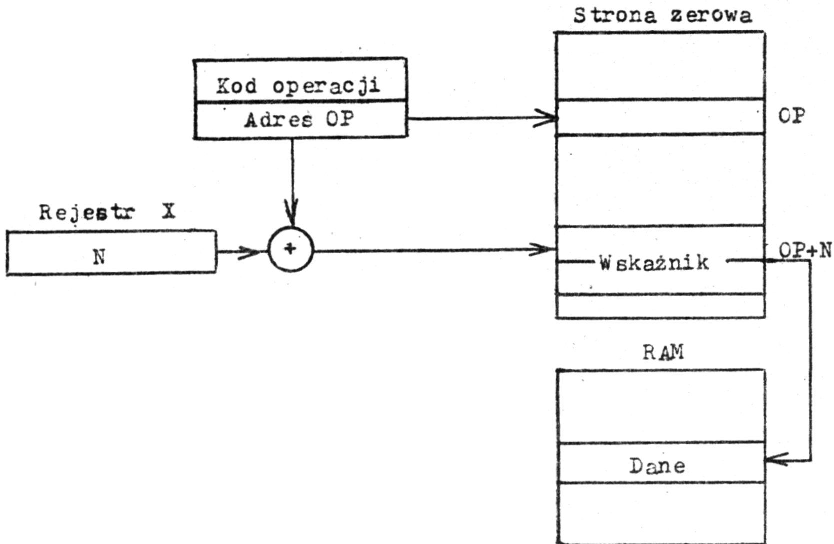
```
00CC  00 30
00CE  40 30
00D0  80 30
```

Pary te można odczytać jako adresy (bajty przedstawione): 3000, 3040, 3080. A oto przykładowa sekwencja rozkazów:

```
LDX #0        X 0
LDA (CC,X)    Ładuje do A zawartość spod adresu 3000
INX           X=1
INX           X=2
EOR (CC,X)    EOR z komórką o adresie 3040
INX           X=3
INX           X=4
STA (CC,X)    Wynik zapisany pod adresem 3080.
```

Jak działa ten program? Na stronie zerowej znajduje się tablica złożona z trzech adresów. LDA (CC,X) pobiera wartość spod pierwszego z nich. Zwiększenie X o 2 powoduje, że EOR (CC,X) wykonywane jest z liczbą, której położenie pośrednio wskazuje następny adres zapisany n a s t r o n i e z e r o w e j. Wreszcie ponowne powtórzenie tego manewru powoduje zapisanie wyniku pod trzecim adresem pośrednio odczytanym ze strony zerowej. Jest to program nieefektywny, to samo można zrealizować trzema rozkazami w trybie absolutnym bez indeksowania. Chodziło jedynie o pokazanie metody obliczania adresów. Poglądowo przedstawia ją rysunek 5.1.

W tym trybie do adresu na stronie zerowej n a j p i e r w dodawana jest zawartość rejestru X, a p o t e m spod adresu na stronie zerowej obliczonego w ten sposób od-



Rys. 5.1 Adresowanie pośrednie preindeksowane X

czytywany jest efektywny adres, czyli adres ostatecznie wskazujący dane. Ponieważ adres na stronie zerowej oblicza się p r z e d dotarciem do komórek z właściwym adresem, tryb ten nosi nazwę preindeksowanego, co można przetłumaczyć jako wcześniej indeksowany czy "przed-indeksowany".

Łatwo zauważyć, że tryb ten ma taką samą wadę, jak omówiony w punkcie 5.5 absolutny wariant: trudno jest na stronie zerowej zmieścić tablicę adresów. Dlatego zdecydowanie szersze zastosowanie znajduje postindeksacja z pomocą rejestru Y.

W obu trybach dostępne są rozkazy, które należą do najważniejszych: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

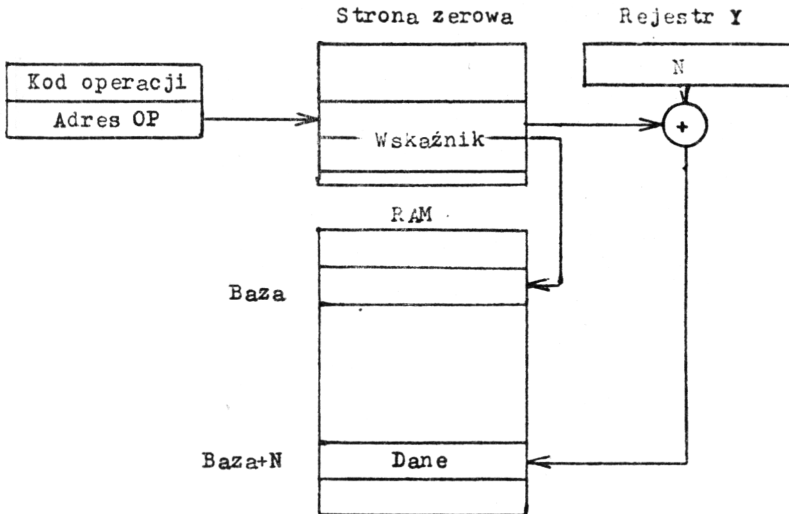
5.8 Z rejestrem Y - postindeksowanie

Nieporównanie użyteczniejszy jest tryb z zastosowaniem rejestru Y, odmienny od poprzedniego i, powiedzmy też sobie przejrzystszy. Jak odbywa się postindeksowanie? W tej meto-

dzie komórki na stronie zerowej mogą zawierać pojedyńcze adresy, ponieważ cała procedura dodawania zawartości Y odbywa się dopiero p o ustaleniu pierwszego, wyjściowego, bazowego adresu obszaru. Dlatego tryb ten nosi nazwę postindeksowanego czyli indeksowanego później, "po". Uwidacznia się to w sposobie zapisywania rozkazów w asemblerze: operand umieszcza się w nawiasie, a Y dopiero za nim. Np.

```
CMP (Z),Y
LDA (20),Y   itd.
```

Rysunek 5.2 wyjaśnia działanie tego trybu.



Rys. 5.2 Adresowanie pośrednie postindeksowane Y

Niechaj znowu na stronie zerowej znajdą się te same, co przedtem trzy adresy:

```
00CC  00 30
00CE  40 30
00D0  80 30
```

Zwróćmy uwagę że wyznaczają one na stronie trzydziestej

odcinki pamięci o długości 40 hex czyli 64 bajtów. Niech obszary te zawierają ciągi po 64 nieduże liczby binarne, które chcemy do siebie dodać i wynik umieścić w 64-bajtowym polu poczynając od adresu 3080 hex. Oto program, który to wykona:

```
     LDY #3F      Zapisujemy w Y długość bloków -1
CYKL CLC          Zawsze przed dodawaniem ...
     LDA (CC),Y   Liczba z pierwszego ciągu w A
     ADC (CE),Y   Dodanie liczby z drugiego cięgu
     STA (DO),Y   Zapisanie wyniku w trzecim ciągu
     BPL CYKL
```

Jak widać, w przeciwieństwie do X, tu rejestr Y pełni poznaną wcześniej rolę licznika pętli. Omówione zastosowanie trybu postindeksowanego ma duże znaczenie wtedy, gdy w chwili pisania programu nie znamy adresów, pod jakimi znajdować się będą dodawane ciągi, a jedynie adresy na stronie zerowej, pod które te adresy będą zapisywane. Może to być użyteczne również wtedy, gdy te same czynności, również bardziej skomplikowane niż przedstawione tu dodawanie, zechcemy wykonywać wobec danych położonych w różnych miejscach pamięci. Wystarczy wówczas wpisać adresy początków odpowiednich pól na stronę zerową i korzystać z nich w wysoce efektywnym trybie adresowania: pośrednim strony zerowej postindeksowanym Y, zwanym też krótko pośrednim indeksowanym Y.

Przypomnijmy dostępne rozkazy: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

Czas wykonania rozkazów w tym trybie jest o cykl dłuższy niż w absolutnym indeksowanym.

Ćwiczenia

x 1. Rozporządzając podanymi adresami można napisać przykładowy program z tego punktu również w trybie absolutnym indeksowanym. Napiszmy go.

2. Obliczmy, ilu cykli wymagać będzie jeden i drugi program.

3. Jakie są podobieństwa, a jakie różnice rozkazów: LDA 10,X i LDA (10),X? Przyjmijmy, że X=8.

5.9 Dodatkowe rozkazy 65C02 i rozkazy "utajone" 6502

Do zmodyfikowanej werdji omawianego w tej książce mikroprocesora 6502, opatrzonej nazwą 65C02 i wykonanej w innej technologii CMOS wprowadzono szereg zmian, które tu pokrótce omówimy. Nowy mikroprocesor zachował zasadnicze parametry 6502. Główna różnica polega na wprowadzeniu kilku nowych rozkazów oraz na zwiększeniu liczby trybów, w jakich dostępne są rozkazy dotychczas istniejące.

Zmiany te nie naruszają dotychczasowej struktury rozkazów i trybów adresowania 6502. Programy opracowane na ten ostatni mikroprocesor można wykonywać na nowym. Oto dokonane uzupełnienia (w nawiasach kody operacji w hex).

Nowe rozkazy:

STZ (store zero) - zastępuje parę rozkazów LDA 0,STA (adres). Dostępny w trybach: strony zerowej (64), strony zerowej indeksowany X (74), absolutny (9C), absolutny indeksowany X(9E).

BPA (branch absolute) - odgałęzienie bezwarunkowe zachowujące pozostałe cechy odgałęzień warunkowych. Tryb względny (80).

TRB (test and reset) oraz TSB (test and set) umożliwiają sprawdzenie bitów w bajcie pamięci i akumulatorze, sygnalizując wynik w znaczniku Z. Dostępne w trybach: absolutnym (TRB - 1C, TSB - 0C) i strony zerowej (TRB - 14, TSB - 04).

Wprowadzono komunikację rejestrów X i Y zes stosem:

PHX (push X on stack - DA) PLX (pull X from stack - FA)
PHY (push Y on stack - 5A) PLY (pull Y from stack - 7A)

Nowe tryby:

Rozkazy 6502 stały się dostępne w dodatkowych trybach.

JMP w pośrednim strony zerowej indeksowanym X (7C);

BIT w natychmiastowym (89), strony zerowej indeksowanym X (34) i absolutnym indeksowanym X (3C);

INC w absolutnym (1A)

DEC w absolutnym (3A).

Wprowadzono nowy tryb pośredni strony zerowej nie indeksowany . Dostępne są w nim: LDA (B2), STA (92), JMP (6C), CMP (D2), ADC (72), SBC (F2), AND (32), ORA (12) i EOR (52).

W absolutnym indeksowanym X zmniejszono o 1 cykl czas wykonania rozkazów ASL, DEC, INC, LSR, ROL i ROR.

Na marginesie tych zmian warto zauważyć, że również w standardowych wersjach mikroprocesora 6502 można z pomocą podanych nowych kodów rozkazów uzyskać wykonanie niektórych czynności nie objętych oficjalną listą rozkazów i nieuwzględnionych w mnemonikach asemblera. Moje próby na Atari wykazały, na przykład, że kod 9C rozkazu STZ wprowadza 1 do wskazanej komórki, a kod 9E niezależnie od wartości w rejestrze zeruje komórkę o adresie o 1 większym niż podany w operandzie. Szereg nowych kodów rozkazów, jeżeli poda się przy nich operandy o właściwej długości, nie powoduje zawieszenia pracy 6502, jakkolwiek nie wydaje się, by przewidziane czynności były wykonywane, albo nie są wykonywane w pełni zgodnie z nowymi rozkazami.

Można to wszystko traktować jako ciekawostki, potwierdzają one jednak słuszność nie od dziś znanej tezy, że nawet twórcy sprzętu nie w pełni znają jego możliwości.

Rozdział 6

NIEKTÓRE TECHNIKI PROGRAMOWANIA

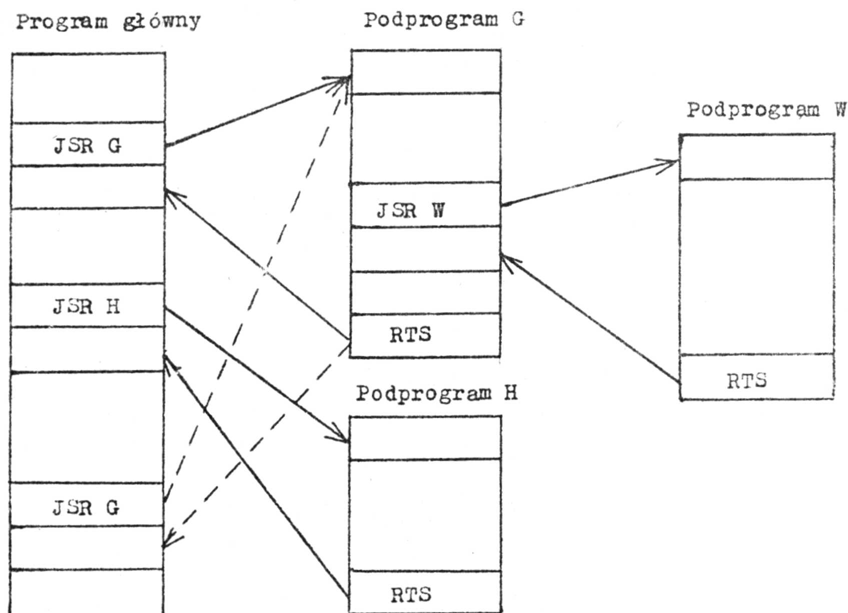
6.1 Podprogramy

Gdy z pewnego ciągu rozkazów możemy wielokrotnie skorzystać w programie, celowe jest wyodrębnienie go w podprogram. Zaletą podprogramu jest to, że z pomocą JSR możemy go wywoływać z dowolnego miejsca w programie i mieć zapewniony z pomocą RTS powrót za każdym razem we właściwe miejsce, to znaczy do rozkazu następującago po JSR. Działanie tego mechanizmu jest takie samo, jak instrukcji Basicu GOSUB i RETURN, tyle że tam adresy początku podprogramu i powrotu określa się numerami linii, a w JM i asemblerze - adresami w programie.

Rozkaz JSR powoduje, że 6502 wstawia na stos adres powrotu, czyli adres następnego rozkazu (pomniejszony dla uproszczenia obliczeń o 1). Z kolei znajdujący się na końcu podprogramu rozkaz powrotu RTS odtwarza poprawne dane, co umożliwia kontynuację programu. Ten sam podprogram można wywoływać z wielu miejsc w programie głównym. Ilustruje to rysunek 6.1.

Możliwe jest również wywoływanie podprogramów z wnętrza innych podprogramów. Korzystanie z podprogramów zapewnia znaczną oszczędność pamięci dzięki skróceniu tekstu programu. Adresy podprogramów umieszczane są przez JSR na stosie w kolejności ich pojawiania się, a zdejmowane ze stosu w kolejności pojawiania się rozkazów RTS. Dzięki temu w przypadku zagnieżdżenia jednych podprogramów w innych następują powroty pod właściwe adresy.

Podprogramy są często tworzone w celu wykonywania cząstkowych obliczeń, których wyniki wykorzystywane są w programie głównym. W takim wypadku pojawia się z jednej strony sprawa przekazania do podprogramu danych wyjściowych, a z



Rys. 6.1 Powiązania programu głównego z podprogramami

drugiej przyjęcia wyników. Czynności te noszą nazwę przekazywania parametrów.

W JM istnieją trzy zasadnicze drogi przekazywania parametrów: za pośrednictwem rejestrów, komórek pamięci oraz stosu.

Warunkiem korzystania do tego celu z rejestrów jest ich dostępność w danej chwili. Taki sposób przekazywania parametrów ma tę zaletę, że nie absorbuje pamięci i uniezależnia od niej podprogram.

Wykorzystanie pamięci do przekazywania parametrów zapewnia większą elastyczność i pozwala przekazywać więcej danych.

Stos jest bardzo wygodnym ogniwem pośrednim przekazywania parametrów z pomocą prostej pary rozkazów PHA i PLA. Ograniczone rozmiary stosu nie pozwalają go przeciążać tym zadaniem. Jest to jednak bez wątpienia najwygodniejsza droga,

zwłaszcza gdy rejestry wewnętrzne są zajęte. W przypadku większych bloków danych eleganckim rozwiązaniem jest posłużenie się w s k a ź n i k a m i (ang. pointers) czyli adresami początków bloków, które można przechowywać w rejestrach, na stosie lub w komórkach strony zerowej, skąd mogą być efektywnie wykorzystywane z pomocą adresowania pośredniego indeksowanego.

Wybór sposobu przekazywania parametrów zależy od programisty. Na ogół dąży się, aby, gdy tylko jest to możliwe, był on jak najbardziej niezależny od pamięci.

Przekazywanie parametrów za pośrednictwem stosu przedstawione będzie na przykładzie podprogramu dzielenia w punkcie 6.3.

6.2 Mnożenie

6502 nie ma rozkazów mnożenia. By wykonać to działanie, trzeba opracować program. Proste mnożenia przez potęgi liczby 2 z pomocą rozkazów przesunięć i obrotów oraz ich kojarzenie z dodawaniem poznaliśmy w punkcie 3.10. Jak uogólnić to na inne liczby?

Mnożenie liczb binarnych przebiega podobnie, jak liczb dziesiętnych. Oto dla porównania przykłady:

114	Mnożna	1011
x 123	Mnożnik	x 101
342		1011
228		0000
114		1011
14022	Iloczyn	110111

Wspólne dla obu metod jest to, że każdy cząstkowy wynik mnożenia przesuwamy o jedną pozycję w lewo i sumujemy te wyniki. Można to osiągnąć również inaczej: przesunąć pierwszy iloczyn cząstkowy o jedną pozycję w prawo, dodać drugi iloczyn cząstkowy, przesunąć sumę o jeden w prawo, dodać trzeci iloczyn, przesunąć sumę itd. Tak właśnie postępuje się w mnożeniu liczb binarnych.

W tych ostatnich zwraca uwagę cecha upraszczająca mnoże-

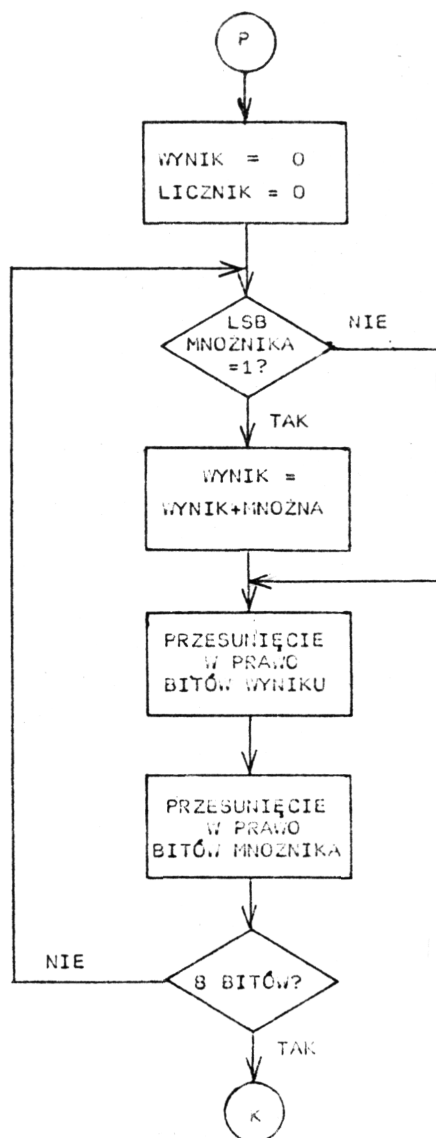
nie: możemy mnożyć tylko przez 1 i 0. Gdy w mnożniku cyfra ma wartość 1, wynik cząstkowy jest po prostu odpowiednio przesuniętą mnożną, a gdy cyfra ma wartość 0,to i wynik cząstkowy = 0. Mnożenie sprowadza się do przesunięć i dodawań.

W mnożeniu dwóch liczb 8-bitowych trzeba osiem razy wykonać przesunięcie w prawo, a także osiem razy sprawdzić wartość kolejnych liczb binarnych mnożnika. Jeżeli cyfra = 1, trzeba dodać mnożną, jeżeli jest ona zerem - opuścić dodawanie. Algorytm tak wykonywanego mnożenia przedstawiony jest na rysunku 6.2.

Przedstawimy poniżej efektywny podprogram mnożenia dwóch liczb 8-bitowych. Ponieważ akumulator A jest jedynym rejestrem wewnętrznym 6502 umożliwiającym przesunięcie bitów, cząstkowe iloczyny gromadzić będziemy w A (górna część) i w komórce o adresie B, najlepiej na stronie zerowej (dolna część). Mnożnik znajdować się będzie pod adresem MK, a mnożna pod adresem MA w pamięci.

```
MNOZ LDA #0      Górna część wyniku inicjalizowana zerem
     STA B       To samo dolna część
     LDX #8      X jest licznikiem przesunięć
CYKL LSR MK      Przesunięcie mnożnika
     BCC NIED    Jeżeli LSB mnożnika = 0, nie ma dodawania
     CLC         Skasowanie przeniesienia
     ADC MA      A=A+MA
NIED ROR A       Obrót A, najniższy bit trafia do C ...
     ROR B       ... i jest "łapany" do górnej części iloczy-
                 nu
     DEX         Zmniejsza licznik o 1
     BNE CYKL    Jeżeli X nie równa się 0, powrót.
     RTS
```

Jak działa ten program? Po inicjalizacji A i B, przeznaczonych na wynik oraz licznika X zaczyna się pętla główna. LSR MK powoduje, że ostatnia w danej chwili cyfra mnożnika trafia do znacznika C. BCC sprawdza C i gdy C=1, n i e wykonuje skoku. Wtedy mnożna jest dodawana do wyniku. Gdy C=0, czynność ta jest pomijana.



Rys. 6.2 Algorytm mnożenia dwóch liczb 8-bitowych

Kolejne rozkaży, ROR A i ROR B wykonują dwie ważne czynności. Po pierwsze, jeżeli było dodawanie ADC MA, to bit przeniesienia wchodzi do najwyższego bitu A. Po drugie, po przesunięciu bitów A w prawo skrajny prawy bit A przechwycony zostaje na najwyższą pozycję w B. Mamy tu ciekawe zjawisko jakby ślizgania się cząstkowych sum w prawo w A i B, tak iż po ośmiu takich przesunięciach A (górna część) i B (dolna część) zawierają iloczyn.

Podprogram mnożenia można włączyć do innego programu i korzystać z niego z pomocą rozkazu skoku do podprogramu JSR. Dlatego na końcu podprogramu znajduje się RTS. Pamiętajmy, że podprogram zostawia iloczyn w A i B. Trzeba go stamtąd zabrać i wykorzystać odpowiednio do realizowanego zadania.

Ćwiczenia

1. Wykonajmy ręcznie mnożenia w postaci binarnej następujących liczb dziesiętnych: a/ 9x9 b/ 3x10 c/ 15x15.

2. Podstawmy do podprogramu MNOZ konkretne adresy i liczby, sprawdźmy wyniki.

6.3 Dzielenie

Liczby binarne dzieli się podobnie jak dziesiętne. Oto przykład dzielenia tej samej pary liczb w układzie dziesiętnym i dwójkowym:

```
  199                   11000111
 ______         ________________
 34427:173      1000011001111011:10101101
 173             10101101
 1712              10111111
 1557              10101101
  1557                100101110
  1557                 10101101
     0                  10101101
                        10101101
                               0
```

Dzielenie binarne jest bardziej pracochłonne, jednak konstrukcyjnie prostsze dzięki temu, że nie wymaga mnożeń innych niż przez 1 i 0. Jest ono czynnością odwrotną do mnożenia. W naszym przykładowym podprogramie będziemy dzielić liczbę 16-bi-

tową przez 8-bitową i zakładac uzyskanie wyniku 8-bitowego. Ponieważ przy zbyt małych dzielnikach wynik nie zmieści się w bajcie, a przez 0 dzielić nie wolno, na początku podprogramu dzielenia możliwości takie zostaną wyeliminowane.

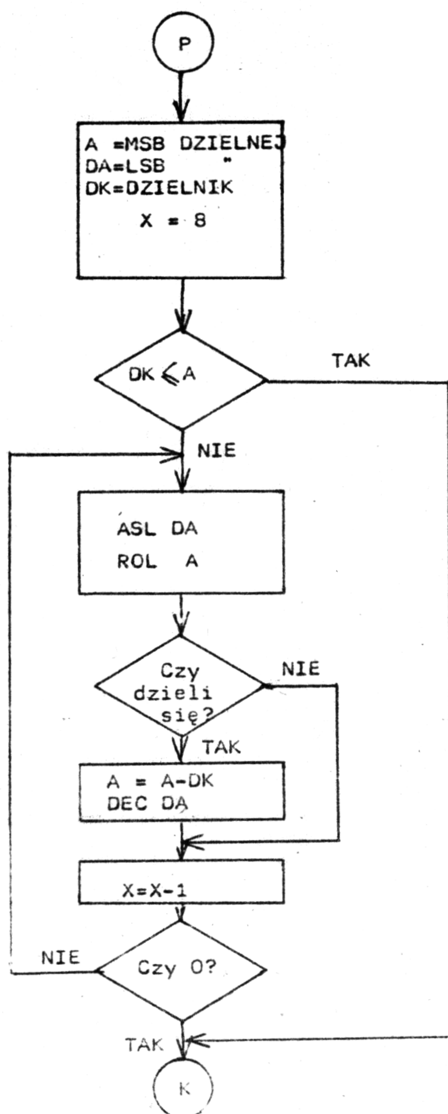
Bardziej znaczący bajt dzielnej umieścimy w akumulatorze. W komórce DA znajdować się będzie mniej znaczący bajt dzielnej, a w komórce DK - dzielnik. Iloraz będzie pozostawiony w DA, reszta w A. Wystarczającym warunkiem poprawnego działania algorytmu jest to, by dzielnik nie przewyższał bardziej znaczącego bajtu dzielnej znajdującego się w akumulatorze.

A oto podprogram dzielenia liczby 16-bitowej przez 8-bitową zrealizowany na podstawie tego algorytmu:

```
DZIEL CMP DK       Jeżeli A nie mniejsze od DK, to
      BCS POWR     skok na koniec programu
      LDX #8       Licznik przesunięć bitów
CYKL  ASL DA       Przesunięcie w LSB dzielnej
      ROL A        i przejęcie najwyższego bitu do A
      CMP DK       Porównanie z dzielnikiem
      BCC OMIN     Jeżeli A <DK, to pomija odejmowanie
      SBC DK       Odejmuje dzielnik od A. C już ustawiony
      INC DA       Wstawia 1 na koniec LSB dzielnej
OMIN  DEX          Licznik zmniejszony o 1
      BNE CYKL     Jeżeli nie zero, powtórzenie czynności
      STA RESZTA   Zapisuje resztę z dzielenia
POWR  RTS          Powrót z podprogramu.
```

Komentarze zdają się dostatecznie wyjaśniać tok obliczeń. Odwrotnie niż przy mnożeniu "ślizg" 16 bitów dzielnej wykonywany jest w lewo. Zwraca uwagę ekonomiczne wykorzystanie młodszego bajtu dzielnej: na opróżniające się od prawej miejsca podprogram wprowadza kolejne bity wyniku, który krok po kroku wypełnia cały ten bajt.

Rozpatrzmy na przykładzie podprogramu DZIEL sprawę przekazywania do niego parametrów z programu głównego i odbioru wyników. Niech parametrami tymi będą konkretne wartości dzielnej i dzielnika, a program odbierze iloraz i resztę, które umieści w komórkach o adresach ILORAZ i RESZTA. Program główny powi-



Rys. 6.3 Algorytm dzielenia liczby 16-bitowej przez 8-bitową

nien zatem przekazać na stos potrzebne dane i ze stosu zdjąć wyniki. Niech ma on następującą postać (dane w hex):

```
DANE LDA #34       Jest to  s t a r s z y  bajt dzielnej
     PHA           Przekazanie go z akumulatora na stos
     LDA #30       Młodszy bajt dzielnej
     PHA           Przekazanie go na stos
     LDA #42       Dzielnik ...
     PHA           ... na stos jako ostatni
     JSR DZIEL1    Skok do podprogramu DZIEL1
     PLA           Odbiór  i l o r a z u
     STA ILORAZ    Zapisanie go w pamięci
     PLA           Odbiór reszty
     STA RESZTA    Zapisanie jej.
```

Należy zwracać uwagę na kolejność przekazywania parametrów na stos i pobierania wyników. Przypomnijmy, że podprogram dzielenia umieszczał dzielnik i młodszy bajt dzielnej w pamięci, a starszy bajt dzielnej w akumulatorze. Parametry pobierane będą przez podprogram ze stosu w kolejności o d w r o t n e j do tej, w jakiej były umieszczane, taka jest bowiem zasada działania stosu. Wygodne jest zatem, by starszy bajt dzielnej był ostatni i pozostał w akumulatorze do dalszego przetwarzania przez podprogram. Po zakończeniu jego wykonania w akumulatorze pozostanie reszta. Najwygodniej będzie, gdy podprogram niezwłocznie, jako pierwszą przekaże ją na stos. Program główny będzie jednak zdejmować resztę ze stosu jako parametr o s t a t n i. O tej zmianie kolejności na odwrotną trzeba pamiętać przy przekazywaniu parametrów i ich odbiorze.

Nasz podprogram DZIEL w jego dotychczasowej postaci nie wykona poprawnie przekazywania parametrów, toteż jego początek i koniec trzeba do tego przystosować. Oto podprogram DZIEL1:

```
DZIEL1 PLA         Pobiera ze stosu  d z i e l n i k
       STA DK      Zapisuje go w komórce o adresie DK
       PLA         Pobiera młodszy bajt dzielnej
       STA DA      Zapisuje go w komórce o adresie DA
       PLA         Pobiera MSB dzielnej i zostawia w A
       CMP DK      Odtąd część obliczeniowa bez zmian
       BCS POWR
       LDX #8
```

```
CYKL  ASL DA
      ROL A
      CMP DK
      BCC OMIN
      SBC DK
      INC DA
OMIN  DEX
      BNE CYKL    Koniec obliczeń
      PHA         Przekazuje na stos  r e s z t ę
      LDA DA      Pod adresem DA jest teraz iloraz
      PHA         Przekazuje go na stos
POWR  LDA #0      Wyjaśnienie w tekście
      PHA
      PHA
      RTS         Powrót do programu głównego.
```

Dlaczego pod etykietą POWR nie można tym razem od razu wracać do programu głównego z pomocą RTS? Przypomnijmy, że nasz podprogram zawiera zabezpieczenie, by dzielnik nie był zbyt mały lub nie równał się zeru. W takim wypadku nie będzie wyników. Program główny jednak o tym "nie wie" i będzie ze stosu po zakończeniu podprogramu pobierać wyniki. Gdy mu czegoś nie damy "na pożarcie", zdejmie ze stosu jakieś inne dane, jeżeli tam są, co w wielu wypadkach może spowodować zawieszenie się komputera bardzo wrażliwego na niepoprawne posługiwanie się stosem.

Wskaźnik stosu S ma tę właściwość, że przy opróżnionym stosie wskazuje jego dół czyli adres 1FF hex, natomiast po próbie pobrania z pustego stosu kolejnego elementu przeskakuje na 100 czyli przybiera wartość taką jak przy stosie całkowicie przepełnionym. Rzadko który program byłby w stanie to przetrwać.

Jest, oczywiście, inny sposób zaradzenia w danym wypadku niespodziankom: sprawdzenie danych już w programie głównym i nie przekazywanie ich do podprogramu, gdy spowodują błędny wynik. Jak to zrobić? Czy coś na tym zyskamy? Odpowiedzi pozostawiam Czytelnikom.

Przedstawione w tym i poprzednim punkcie podprogramy mnożenia i dzielenia pokazują wykonywanie tych działań, które można stosować do dowolnie dużych liczb. Powstało i nadal

powstaje wiele ciekawych algorytmów w celu realizacji tego zadania, np. dla bardzo wielkich liczb.

Ćwiczenia

x 1. Wykonajmy na papierze dzielenia na następujących liczbach binarnych: a/ 1000110:111 b/ 1000000:1010 c/ 11001000:1101.

2. W ćwiczeniu 1 wyraźmy liczby w postaci dziesiętnej.

3. Podstawmy do podprogramu DZIEL konkretne liczby i adresy, sprawdźmy wyniki.

4. Wykonajmy to samo za pośrednictwem programu DANE.

6.4 Porównania liczb dwubajtowych

Liczby odejmujemy od prawej, ale porównujemy od lewej. Już pierwsza cyfra 29 i 30 pozwala ustalić, która z tych liczb jest większa. Podobnie postępujemy przy porównywaniu wartości dwu lub wielobajtowych: zaczynamy od najstarszego bajtu.

Przypuśćmy, że dwie liczby 16-bitowe znajdują się pod adresami P i Q z młodszymi bajtami jako pierwszymi. Poniższy program umożliwi ustalenie, czy P jest mniejsze od Q:

```
         LDA P+1      Bardziej znaczący bajt P w A
         CMP Q+1      Porównanie z MSB Q, co odpowiada odję-
                      ciu Q
         BNE RELACJA  Skok, jeżeli wartości nierówne
         LDA P        W przypadku równości badanie LSB P ...
         CMP Q        ... i Q
RELACJA  BCC MNIEJSZA Miejsce sprawdzania relacji.
```

Jeżeli porównanie starszych bajtów nie potwierdzi ich równości, dalsze sprawdzanie przestaje być potrzebne i od razu następuje przeskok do etykiety relacja. Jeżeli znacznik C jest skasowany (BNE go nie zmienia), oznacza to, że P jest mniejsze od Q, co ujawni odgałęzienie BCC. W przypadku równości pierwszych bajtów następuje porównanie drugich i jego wynik również odzwierciedlają znaczniki.

Stosując przy etykiecie inne rozkazy można wykonywać odgałęzienia przy odmiennych relacjach P i Q. Np. zastąpienie BCC MNIEJSZA przez BCS WR spowoduje skok, gdy P jest większe

lub równe Q. Zamieniając liczby miejscami relacje P i Q możemy odwrócić z pomocą analogicznego postępowania. Test równości wykona rozkaz BEQ.

Analogiczne porównania możliwe są również z pomocą rejestrów X i Y oraz rozkazów CPX i CPY.

6.5 Liczby ze znakiem

Mowa już była o trudnościach powstających przy korzystaniu z liczb ze znakiem. Przyczyną tych trudności jest nie tylko zajęcie najwyższego bitu przez znak, lecz także to, że pozostała część wyrażona jest w kodzie uzupełnieniowym. Szczegółowe omówienie tej rozległej problematyki nie jest możliwe w ramach niniejszej książki. Ograniczymy się do kilku informacji.

P o r ó w n a n i a wymagają wykorzystania informacji dostarczanej przez znacznik nadmiaru V. Rozpatrzmy to na przykładzie liczb 8-bitowych. Jeżeli P=-29, a Q=100, to P-Q stosowane w porównaniu powinno dać -129. W rzeczywistości jednak powstanie wynik pozornie dodatni, ponieważ liczba nie zmieściła się na 7 bitach i skasowała bit znaku. Znacznik V sygnalizuje sytuację, gdy wynikiem odejmowania jest powstanie liczby większej niż 127 lub mniejszej niż -128. Przypomnijmy, że rozkazy porównań nie zmieniają tego znacznika, trzeba więc zastosować odejmowanie. W obu sytuacjach nadmiaru wynik odejmowania jest poprawny, a jedynie znak jest odwrotny. Toteż tam, gdzie zastosowalibyśmy dla sprawdzenia warunku BMI, można zastosować wówczas BPL. Oto program, który w przypadku, gdy P jest mniejsze od Q powoduje skok do etykiety MNIEJSZA:

```
        LDA P
        SEC
        SBC Q           Odejmowanie P-Q
        BVS JEST.V      Skok, gdy jest nadmiar
        BPL DALEJ       Przejście dalej, gdy P> Q
        BMI MNIEJSZA    Przejście pod MNIEJSZA, gdy P< Q
JEST.V  BPL MNIEJSZA    Przejście tamże po sprawdzeniu znaku
DALEJ   (następny rozkaz)
```

Efektem zastosowania aż czterech rozkazów odgałęzień będzie wyodrębnienie sytuacji, gdy P jest mniejsze od Q. Po raz pierwszy poznajemy tu zastosowanie BVS.

Porównanie liczb 16-bitowych J i L wymaga zastosowania opisanej przed chwilą procedury tylko do ich MSB czyli J+1 i L+1, ponieważ w młodszych bajtach najwyższy bit nie jest bitem znaku i trzeba tu zastosować takie samo porównanie, jak dla liczb bez znaku.

Dodawanie i odejmowanie dwóch liczb 8-bitowych ze znakiem odbywa się podobnie, jak w przypadku takich liczb bez znaku. Podobnie również (wyjaśnienia z braku miejsca pominiemy) liczby 16-bitowe ze znakiem można dodawać i odejmować jak analogiczne liczby bez znaku.

Natomiast mnożenie liczb ze znakiem przebiega inaczej niż w przypadku liczb bez znaku. Rozpatrzmy to na przykładzie dwóch liczb 8-bitowych. Jeżeli mnożymy T przez -U, to w rzeczywistości mnożymy T przez 256-U bez znaku i uzyskujemy 256xT-TxU. Tymczasem reprezentacją 16-bitowego wyniku w kodzie uzupełnieniowym powinno być 256^2-TxU.

W.D. Maurer [3] następująco przedstawia algorytm mnożenia liczb ze znakiem T i U:

1. Ustalić c równe EOR znaków liczb T i U.
2. Ustalić T'=T. Jeżeli T' jest ujemne, to T'=-T'.
Ustalić U'=U. Jeżeli U' jest ujemne, to U'=-U'.
4. Pomnożyć T' przez U' uzyskując W'.
5. Jeżeli c=0, to W' jest wynikiem. Jeżeli c=1, to wynikiem jest -W'.

Otrzymanie negacji 8 i 16-bitowych liczb ze znakiem uzyskuje się tak samo jak w przypadku liczb bez znaku: należy wszystkie bity jedynkowe zamienić na zerowe i odwrotnie.

Analogiczny algorytm można zastosować przy dzieleniu liczb ze znakiem.

Ćwiczenia

1. Jaka jest najmniejsza, a jaka największa 16-bitowa liczba całkowita ze znakiem wyrażona w postaci dziesiętnej?

6.6 Tablice

Tablice liczb binarnych rozmaitej długości stanowią jednę z najszerzej stosowanych struktur danych. W Basicu tworzenie tablicy polega na zadeklarowaniu jej charakteru i rozmiaru. DIM A$(40), A(10) tworzy złożoną z 40 1-bajtowych komórek zmienną indeksowaną A$, do której wprowadzać możemy znaki, a także 10-elementową tablicę wartości liczbowych, na którą np. Atari rezerwuje aż 60 bajtów - po 6 na liczbę zmiennopozycyjną.

Do poszczególnych elementów zmiennych indeksowanych docieramy z pomocą ich indeksów. A$(3) oznacza trzeci element zmiennej znakowej A$, A(7)pozwala uzyskać dostęp do siódmego elementu zmiennej A. W Atari indeksy zmiennych znakowych zaczynają się od 1, a liczbowych od 0.

W asemblerze nie ma takich form deklarowania tablic i uzyskiwania dostępu do ich elementów. Można jednak utworzyć tablicę nadając etykietę czyli nazwę a d r e s o w i zerowego elementu tablicy. Jeżeli nazwiemy ją T, to odpowiednikiem T$(6) w Basicu będzie komórka o adresie T+6. Posługiwanie się elementami tablic ułatwiają tryby adresowania indeksowanego. Jeżeli chcemy zawartość T(6) załadować do akumulatora, wykonają to rozkazy:

```
LDX #6          lub          LDY #6
LDA T,X                      LDA T,Y
```

W JM często posługujemy się danymi d w u b a j t o w y m i, toteż nieraz celowe jest tworzenie tablic takich danych np. adresów podprogramów. W takim wypadku przesunięcie adresu w stosunku do początku tablicy będzie dwukrotnie większe nim indeks elementu. Przypuśćmy, że chcemy ustalić, czy element T(J) tablicy jest większy niż element T(K). Posługując się metodą poznaną w punkcie 6.4 wykonujemy to następująco:

```
LDA J       Indeks pierwszego elementu
ASL A       Mnożymy przez 2
TAX         Przenosimy do rejestru X
LDA K       Indeks drugiego elementu
ASL A       Mnożymy ...
```

```
         TAY             ... i przenosimy do rejestru Y
         LDA T+1,X       MSB T(J)
         CMP T+1,Y       Porównanie z MSB T(K)
         BNE RELACJA     Jeżeli nie równe, koniec porównania
         LDA T,X         Jeżeli MSB równe, porównujemy LSB
         CMP T,Y
RELACJA  BCC MNIEJSZA
```

Pod etykietą RELACJA można, jak wspominaliśmy, stosować różne rozkazy odgałęzień.

Przedstawione dotychczas metody organizowania dostępu do tablic jedno- i dwubajtowych ograniczają ich rozmiar do 256 bajtów, taki jest bowiem zasięg 8-bitowych rejestrów X i Y. W przypadku dłuższych tablic do dwubajtowego adresu początku tablicy trzeba dodawać dwubajtowy indeks. Najwygodniej jest posłużyć się w tym celu w s k a ź n i k i e m, czyli adresem umieszczonym na stronie zerowej i adresowaniem indeksowanym.

Poniższy program pokazuje metodę uzyskania dostępu do elementu T(J) długiej tablicy. Użyte tu zostało stosowane w asemblerze oznaczenie mniej znaczącego bajtu adresu T oraz bardziej znaczącego >T.

```
LDA <T          Dodajemy mniej znaczące bajty adresu i J
CLC
ADC J
STA ZP          Tworzą one LSB wskaźnika na stronie zero-
                wej ZP
LDA >T          Dodajemy bardziej znaczące bajty
ADC J+1
STA ZP+1        Tworzą one MSB wskaźnika ZP
LDY #0          Posługujemy się adresowaniem pośrednim,
                Y=0
LDA (ZP),Y
```

W danym wypadku adres T(J) znalazł się na stronie zerowej i dlatego w rejestrze Y umieściliśmy 0. Program można nieco usprawnić stosując godny poznania chwyt. Na stronie zerowej w komórce ZP umieszczony 0, a w ZP+1 bardziej znaczącą

część adresu. Wówczas do rejestru Y wprowadzamy LSB adresu. Razem tworzą one poprawny adres. W tym celu w podanym tu programie należy STA ZP zastąpić przez TAY i usunąć rozkaz LDY #0. Zaoszczędzamy trzy bajty miejsca i trzy cykle zegara.

Ćwiczenia

1. Napiszmy w asemblerze równoważniki następujących stwierdzeń Basicu: a/ F=T(G) x b/ T(6)=K c/ T(N)= T(3)

2. Napiszmy w Basicu równoważniki następujących ciągów rozkazów asemblera:

```
x a/ LDX J          b/ LDA T+1         x c/ LDX M
     LDA T,X           STA N                DEC U,X
     STA U,X
```

6.7 Przemieszczanie dużych bloków pamięci

W punkcie 5.4 podaliśmy przykład programu kopiującego blok pamięci w inne jej miejsce. Program ten umożliwiał jednak przemieszczanie najwyżej 256 bajtów. Tymczasem często trzeba przemieszczać większe bloki. Często np. w podprogramach służących do tworzenia zbioru polskich znaków na Atari stosuje się metodę przemieszczania 1024 bajtów danych standardowego zbioru znaków z ROM do RAM w celu ich przerobienia. Jekkolwiek istnieje wygodniejsza metoda stworzenia zbioru polskich znaków w drodze zamiany ROM na RAM pod tymi samymi adresami, to jednak nieraz przydatne mogą być takie przemieszczenia bloków dłuższych niż 256 bajtów. Ktokolwiek wykonywał tego rodzaju czynność z pomocą PEEK i POKE w Basicu, wie, ile czasu zabiera takie przemieszczenie bloków danych. Znacznie szybciej wykona to podprogram w JM.

Powiedzmy, że z programu głównego przekazujemy za pośrednictwem stosu niezbędne dane: adresy początku i końca przemieszczanego bloku oraz adres początkowy bloku, do którego chcemy przemieścić dane. W ostatnim bajcie prześlemy do podprogramu liczbę przekazywanych parametrów. W rozdz. 8 wyjaśnimy, dlaczego jest to stosowane.

Oto możliwy kształt takiego programu z przykładowo podanymi granicami bloków. Użyliśmy tu adresów, między którymi

rzeczywiście rozciąga się w ROM Atari drugi, międzynarodowy zbiór znaków, oraz dowolnie wybranego docelowego adresu 4000 hex.

Oto program, który przekazuje takie parametry (dane w hex) :

```
ZNAKI LDA #0
      PHA
      LDA #CC
      PHA          Adres początku zbioru znaków - CC00
      LDA #FF
      PHA
      LDA #CF
      PHA          Adres końca zbioru znaków - CFFF
      LDA #0
      PHA
      LDA #40
      PHA          Adres początku obszaru docelowego 4000
      LDA #3
      PHA          Liczba parametrów
      JSR MOVE
      .END
```

Przekazaliśmy parametry z młodszymi bajtami jako pierwszymi. Końcowe .END to komenda asemblera sygnalizująca koniec programu.

W podprogramie potrzebne będą nieco inne dane. Początki bloków źródłowego SKAD i docelowego CEL oraz liczba 256-bajtowych przemieszczanych bloków i ewentualna końcówka. W tym wypadku kilobajt dzieli się bez reszty na bloki o rozmiarach strony pamięci. Czasem jednak końcówka zostaje i przeznaczymy dla niej dodatkowy bajt. Oba adresy umieścimy na stronie zerowej w bezpiecznym na Atari obszarze CB-D1 hex. Oto podprogram :

```
MOVE PLA          C
     CMP #3       Czy 3 parametry?
     BNE POWR     Jeżeli nie - wyjście z programu
     PLA
```

```
     STA ZC+1    Starszy bajt wskaźnika na stronie  z e r o-
                 w e j
     PLA
     STA ZC      Młodszy bajt tego wskaźnika
     PLA
     STA TEMP+1  Adres końca obszaru źródłowego ...
     PLA         czasowo przechowujemy pod adresem TEMP
     STA TEMP
     PLA         Starszy bajt wskaźnika początku obszaru
     STA ZZ+1    źródłowego zapisujemy na stronie zerowej
     PLA
     STA ZZ      To samo młodszy bajt
     LDA TEMP
     SEC
     SBC ZZ      Różnicę młodszych bajtów początku i końca
     STA RESZTA  obszaru źródłowego zapisujemy (tu 0)
     LDA TEMP+1  Obliczamy liczbę pełnych 256-bajtowych
     SBC ZZ+1    bloków ...
     TAX         i przenosimy do rejestru X
     LDY#0       Wyzerowanie rejestru Y
CYKL LDA (ZZ),Y  Czyta z obszaru źródłowego ...
     STA (ZC),Y  i zapisuje w docelowym
     DEY         Następny znak
     BNE CYKL    Gdy nie koniec 256-bajtowego bloku - cykl
BLOK INC ZZ+1    Zwiększa o 1 starsze bajty adresów ...
     INC ZC+1    obszaru źródłowego i docelowego
     DEX         Zmniejsza licznik bloków 256-bajtowych
     BMI KONC
     BNE CYKL
     LDY RESZTA  By przemieścić końcówkę
     BNE CYKL
KONC RTS         Powrót do programu.
```

Program wymaga komentarza. Aż 23 rozkazy zajęło rozmieszczenie wskaźników początków obszarów źródłowego ZZ i docelowego ZC na stronie zerowej, wyliczenie liczby bloków 256-bajtowych, której licznik umieściliśmy w X, oraz liczby pozostałych bajtów do przemieszczenia, którą umieściliśmy pod adre-

sem RESZTA.

Dalszych 10 rozkazów składa się na zasadniczą część programu. Uwagi wymagają cztery rozkazy od etykiety CYKL. Za pierwszym razem przeniesiony zostanie bajt zerowy obszaru przemieszczanego. Potem jednak z m n i e j s z y m y rejestr Y. Oznacza to, że jako następny przeniesiony będzie bajt 255 obszaru, a potem niższe. To zakłócenie kolejności przenoszenia służy osiągnięciu większej zwartości programu.

Gdy przeniesionych będzie 256 znaków, oba wskaźniki na stronie zerowej zwiększamy o 256– starszy bajt wzrasta O 1 . Licznik w rejestrze X zostaje zmniejszony.

Drugim ciekawym miejscem w programie jest para rozkazów:

```
BMI KONC
BNE NEXT
```

Pierwszy rozkaz nie spowoduje odgałęzienia na koniec, gdy w X będą wartości 3, 2, 1, 0. Będzie działać niby otwarta bramka przepuszczająca program do następnego rozkazu. BNE CYKL powodować będzie powrót na początek pętli, gdy w X będą wartości 3, 2, 1. Przy X=0 i ta bramka przepuści program dalej. Tu do rejestru Y wprowadzona będzie liczba pozostałych bajtów do przeniesienia i program wróci do etykiety CYKL, by je wykonać.

Nastąpi wreszcie znamienny moment. W rejestrze X jest 0. DEX zmniejsza X czyli nadaje mu wartość ØFF - 255 dec . I oto teraz ciągle przedtem otwarta bramka BMI KONC z a m y k a s i ę, bowiem ØFF u s t a w i ł o znacznik wyniku ujemnego N i BMI odczytuje tę wartość jako ujemną. Choć nie myśleliśmy tu ani chwili o liczbach ujemnych, BMI pomogło na dobre wyjść z podprogramu po wykonaniu całego zadania.

Czy nie przypomina to ciekawej łamigłówki? Ukazuje zarazem, jak zróżnicowaną rolę mogą pełnić rozkazy odgałęzień i jak przez ich trafny dobór można osiągnąć program zwarty i wysoce sprawny. Pomysł takiego układu rozkazów zaczerpnęliśmy z książki Rodneya Zaksa [4].

6.8 Ujemne indeksowanie

A oto kolejny ciekawy pomysł opisany przez W.D. Maurera [3].

W poprzednim punkcie zwracała uwagę dość dziwna kolejność dostępu do bajtów pamięci. Zdarza się jednak nieraz, że - na przykład, porównując bloki pamięci - chcemy wykonywać tę czynność w kolejności rosnących adresów, a nie odwrotnie. W takim wypadku, gdy indeks zaczynamy, powiedzmy, od 0, a chcemy dojść do jego wartości 6, musimy za każdym obrotem pętli sprawdzać, czy indeks osiągnął tę wartość. Czy można tego uniknąć? Tak. Drogą do tego jest tzw. ujemne indeksowanie. Polega ono na tym, że docelową wartość indeksu, n, np. w omawianym przykładzie 6, zastępujemy wartością w kodzie uzupełnieniowym -n czyli wyrażając to liczbami bez znaku 256-n . W naszym przypadku będzie to 250.

Zwiększając tę wartość sześciokrotnie, dojdziemy do zera i nie będzie potrzebne porównywanie, bo znacznik Z zostanie ustawiony.

Rozpatrzmy to na przykładzie niedużego programu, który w 64-bajtowej tablicy T poszukuje pierwszego wystąpienia bajtu o wartości W, czyli wymaga posuwania się naprzód od początku tablicy. Oto program - dane w hex - w wersji "tradycyjnej":

```
     LDX #0         Zaczynamy od bajtu 0 tablicy
     LDA W          Wartość poszukiwana stale w A
CYKL CMP T,X        Porównanie z W
     BEQ ZNAL       Jeżeli T(X)=W, to skok do "znalezione"
     INX            Do następnego bajtu T
     CPX  40        Czy ostatni?
     BNE CYKL       Jeżeli nie, wznawiamy porównywanie
     .END
```

A oto ten sam program z zastosowaniem ujemnego indeksowania - liczby także w hex :

```
     LDX  100-40
     LDA W
CYKL CMP T-C0,X     Odejmujemy  100-40  czyli C0
     BEQ ZNAL
     INX
     BNE CYKL
     .END
```

Zwróćmy uwagę, że w pętli są obecnie 4 rozkazy, a nie jak przedtem 5. Przestało być potrzebne CPX. Jest to podstawowa zaleta indeksowania ujemnego. Nie można go jednak stosować, gdy liczbę kroków pętli określa zmienna. Łączy się z nim także większe niebezpieczeństwo błędów. Dlatego warto je stosować przede wszystkim wtedy, gdy zależy nam na szybkości wykonania.

6.9 Modyfikacja adresów

Jedną z ciekawych metod programowania, w wielu wypadków pozwalającą udoskonalić program, jest metoda zwana modyfikacją adresów. Chodzi w niej o to, że w trakcie wykonywania programu zmieniamy w nim jeden z operandów adresowych.

Można to zilustrować prostym programem, który zapiszemy w kodzie maszynowym i asemblerze dane w hex .

```
0600 A9 00                   LDA #0
0602 8D 10 06           CYKL STA 610
0605 EE 03 06                INC 603
0608 D0 F8                   BNE CYKL
```

Co powoduje tu rozkaz INC 603? Zmienia on adres w rozkazie STA 610 na STA 611 i w efekcie zero z akumulatora zostanie wprowadzone do następnej komórki. Dziać się tak będzie, dopóki w komórce 603 nie pojawi się zero.

Przykład ten jest o tyle nietrafnym zastosowaniem modyfikacji adresów, że ten sam cel możemy osiągnąć prościej z pomocą adresowania indeksowanego:

```
0600 A9 00                   LDA #0
0602 A8                      TAY
0603 99 10 06           CYKL STA 610,Y
0604 C8                      INY
0605 D0 FC                   BNE CYKL
```

Program jest krótszy i oszczędzamy na każdym obiegu pętli 3 cykle zegarowe.

Wiele jest jednak sytuacji, w których modyfikacja adresów pozwala znacznie ulepszyć program. Spośród niezliczonych jej zastosowań wybierzmy dwa przykłady zaczerpnięte z W.D.

Maurera [3] i krótko jedynie skomentowane.

Podprogram LELA (load element of long array - załaduj do A element długiej tablicy) operuje na tablicy o nazwie ARRAY. Oto jego zapis w kodzie maszynowym i asemblerze:

```
0600 AD FE 20 LELA        LDA ARRAY
0603 EE 01 06             INC LELA+1
0606 D0 03                BNE LELA1
0608 EE 02 06             INC LELA+2
060B                LELA1 RTS
```

Przy pierwszym wywołaniu podprogram załaduje do A początkowy element tablicy ARRAY, po czym poznaną już metodą zwiększy w pierwszym rozkazie operand adresowy o 1. Ponieważ będzie to FF, nastąpi przeskok do końcowego RTS i powrót do programu głównego.

Przy następnym wywołaniu podprogramu w komórkach jego pierwszego rozkazu będzie już inny operand adresowy.20FF. Spod tego adresu podprogram pobierze wartość, zwiększy o 1 liczbę pod adresem 601, która przybierze wartość 0. Bramka BNE będzie otwarta i zostanie wykonany następny rozkaz zwiększający o 1 wartość w komórce 602 czyli starszy bajt operandu adresowego, po czym nastąpi wyjście z podprogramu. Jaki będzie teraz adres w operandzie pierwszego rozkazu? 2100 - czyli znowu adres następnej komórki w tablicy ARRAY.

Tak więc za każdym wywołaniem treść podprogramu LELA ulegać będzie zmianie w operandzie pierwszego rozkazu i za każdym razem podprogram pobierać będzie dane z następnej komórki tablicy ARRAY. Wiele jest sytuacji, w których można to użytecznie wykorzystać.

Jeżeli zdefiniowaliśmy wcześniej adres ARRAY, to przywrócenie początkowego stanu tablicy można osiągnąć z pomocą sekwencji rozkazów:

```
LDA <ARRAY         LSB adresu tablicy
STA LELA+1
LDA >ARRAY         MSB adresu tablicy
STA LELA+2
```

Drugi przykład dotyczy modyfikacji adresu względnego w

rozkazach odgałęzień. Załóżmy, że chcemy w asemblerze zrealizować konstrukcję analogiczną do często stosowanej w Basicu:

ON K GOTO 100, 200, 300, 400, 500

Przypomnijmy, że gdy K=1, następuje skok do linii 100, gdy K=2 - do linii 200 itd. Niech w asemblerze numerom linii odpowiadają etykiety L1, L2, L3, L4, L5. Przypuśćmy, że K znajduje się w rejestrze X. Wówczas program realizujący odgałęzienia odpowiednio do wartości K od 1 do 5 można zapisać następująco:

```
DEX          Powoduje, że gdy K było równe 1 ...
BEQ L1       następuje skok do L1
DEX          Gdy K było równe 2 ...
BEQ L2       następuje skok do L2 itd.
DEX
BEQ L3
DEX
BEQ L4       Skok do L4
BNE L5       Gdy K ma wartość inną, skok do L5
```

Zakłada się, że K rzeczywiście przybiera wartości od 1 do 5. Program zajmuje 14 bajtów, a czas jego wykonania zależnie od wartości K wynosi 5-19 cykli, przeciętnie 12.6 cykla.

W celu zastosowania modyfikacji adresu względnego należy wbudować w program rozkaz o postaci:

```
MODYF BNE 00
```

oraz zastępować w nim bajt adresowy wartościami etykiet L1-L5. Najprościej jest to uczynić budując 5-bajtową tablicę wartości tych etykiet umieszczoną pod adresem ETYKIETY. Jeżeli następny rozkaz za rozkazem modyfikowanym ma etykietę MOD1, to dla uzyskania efektywnego adresu względnego należy odjąć go od adresów, pod które następują skoki w naszym programie. Wówczas tablica przedstawia się następująco:

```
ETYKIETY .BYTE L1-MOD1, L2-MOD2, L3-MOD1, L4-MOD1, L5-MOD1
```

Komenda asemblera .BYTE powoduje, że następne wartości umieszczane są w kolejnych bajtach. Jeżeli nie mamy asemblera, odpowiednie adresy względne można obliczyć ręcznie.

Po utworzeniu tej tablicy implementacja ON ... GOTO z zastosowaniem modyfikacji adresów przedstawiałaby się następująco:

```
      LDA ETYKIETY-1,X    Załadowanie adresu względnego z tab-
                          licy
      STA MODYF+1         Wpisanie go do operandu BNE
MODYF BNE 00
MOD1  (następny rozkaz)
```

Opisana realizacja konstrukcji ON ... GOTO wymaga 8 bajtów na rozkazy i 5 - na tablicę. Czas trwania wynosi 11 cykli, a gdy BNE nie jest wykonywane 10 cykli.

Czasami odległość skoku, jaką zapewniają odgałęzienia, może nie wystarczyć. Rozwiązanie problemu omówimy w następnym punkcie.

Poznaliśmy dwa przykłady modyfikacji adresów. Mimo widocznych zalet metoda ta ma pewne ograniczenia. W pierwszym przykładzie przed zastosowaniem podprogramu LELA należy pamiętać zawsze o nadaniu adresowi ARRAY początkowej wartości.

Drugie ograniczenie dotyczy wszelkich metod modyfikacji adresów: można je oczywiście, stosować tylko wobec programów zapisanych w RAM.

<u>Ćwiczenia</u>

x 1. Program LELA można napisać inaczej zastępując dwa pierwsze rozkazy następującymi:

```
LELA LDA ARRAY,Y
     INY
```

W programie inicjalizacji należy wówczas pierwszy wiersz zastąpić przez:

```
LDA #0
TAY
```

W wariancie tym komórka LELA+1 nie jest modyfikowana.

Jakie są jego zalety i wady?

x 2. W rejestrze X znajduje się liczba w granicach od 0 do 7. Co wykonuje następujący program?:

```
      STX SKOK+1
      LDA Q
SKOK  BNE 0
      LSR
      LSR
      LSR
      LSR
      LSR
      LSR
      LSR
      STA Q
```

6.10 Dalekie skoki warunkowe

Rozkazy odgałęzień mają ograniczony zasięg skoku. Można pokonać tę trudność przez powiązanie ich ze skokiem bezwarunkowym JMP pozwalającym dotrzeć do każdego adresu. Umożliwia to następująca konstrukcja:

```
     BNE POSR       Skok warunkowy do adresu pośredniego
. . . . . . .        dalsze rozkazy
POSR JMP CEL        Skok do adresu docelowego  naprzód lub
                    wstecz
```

Jeżeli adres pośredni POSR znajdzie się jeszcze w obrębie programu, musi on przeskakiwać nad trzema bajtami zajętymi przez JMP CEL. Na przykład:

```
. . . . . . .        poprzednie rozkazy
     JMP DALEJ       Przeskok
     JMP CEL         Nasz skok
DALEJ  dalsze rozkazy programu
```

Adres POSR jest tu jakby trampoliną umożliwiającą dalszy skok.

6.11 Modyfikacja danych w rozkazach w trybie natychmiastowym

Poza opisaną wcześniej modyfikacją adresów można w asemblerze stosować również modyfikację danych zawartych w operandzie rozkazów w trybie natychmiastowym.

Rozpatrzmy to na przykładzie programu wykonującego z pomocą jedynie przesunięć bitów i dodawania mnożenie liczby w akumulatorze przez 10. Chodzi tu o mnożenie niedużych liczb w granicach 0-25.

```
     ASL A        2xA
     STA TEMP     Zapisujemy w komórce tymczasowej
     ASL A        4xA
     ASL A        8xA
     CLC
     ADC TEMP     8xA+2xA=10xA
```

A teraz wersja z modyfikacją danych:

```
     ASL A
     STA DODA+1
     ASL A
     ASL A
     CLC
DODA ADC #0
```

Co się zmieniło? Rozkaz w drugim wierszu zamiast zapisywać wynik w komórce tymczasowej wprowadza go do komórki DODA+1, która jest *operandem* ostatniego rozkazu. Teraz rozkaz ten jest w trybie natychmiastowym. Oszczędziliśmy bajt na tym rozkazie oraz bajt na niepotrzebnym TEMP i dwa cykle r czasie wykonania, jeżeli TEMP nie był na stronie zerowej.

Opisana modyfikacja rzadko okazuje się bardziej efektywna niż posługiwanie się danymi na stronie zerowej, jeżeli ta nie jest przeciążona innymi zadaniami. Wyjątkiem są sytuacje, gdy ADC lub inny rozkaz dostępny w trybie natychmiastowym stosujemy w pętlach.

<u>Ćwiczenia</u>

x 1. W podprogramie MOVE w punkcie 6.7 użyliśmy dwubajtowej stałej pośredniej TEMP. Jak zastąpić to zastosowaniem opi-

sanej modyfikacji danych?

2. Czy osiągniemy zysk: a/ czasu wykonania, b/ miejsca, jeżeli TEMP był na stronie zerowej?

Rozdział 7

Z BASICU W JĘZYK MASZYNOWY

7.1 Na granicy dwóch języków

Jeżeli ktokolwiek zechce przejrzeć pewną liczbę programów napisanych w Basicu, przekona się, że w bardzo znacznej ich części obecny jest język maszynowy. Potwierdza to, że niektóre zadania można skuteczniej rozwiązać pisząc odpowiednie podprogramy w asemblerze. Złudzeniem jest zresztą, że w jakimkolwiek języku wysokiego poziomu programiści zdolni byliby zrealizować każde z bogatej palety zadań możliwych do wykonania z pomocą komputera. Natomiast każde dostępne dla niego zadanie można wykonać w języku maszynowym. Te, którym nie podoła, okażą się po prostu niemożliwe do realizacji na komputerze.

W poszczególnych komputerach opartych na 6502 stosuje się dość zróżnicowane formy współpracy Basicu z językiem maszynowym. Dlatego po raz pierwszy stykamy się tu z problematyką, której omówienie celowe jest powiązać z konkretnym komputerem, a mianowicie z Atari. Mimo różnic widoczna jest zasadnicza cecha wspólna: bez względu na komputer podprogram w JM opracowujemy przecież w języku tego samego mikroprocesora i w oparciu o jego listę rozkazów. Różnice polegają natomiast na sposobie wiązania dwóch języków, a ściślej włączania podprogramów w JM do programów w Basicu. W Atari służy do tego przede wszystkim instrukcja czy, jak się ją często określa, f u n k c j a, USR (user subroutine - podprogram użytkownika).

7.2 Drogi komunikacji

USR odgrywa podstawową rolę w uruchamianiu podprogramu w JM i zapewnianiu automatycznego powrotu z niego do Basicu, a także w przekazywaniu parametrów w obu kierunkach. Wiązanie dwóch języków jest na Atari nieraz korzystne również dla-

tego, że komputer ten ma dobrze rozwiązany edytor Basicu, który odciąża programującego od części wysiłków związanych z obsługą obrazu ekranowego. Natomiast język maszynowy góruje nad Basikiem wszędzie tam, gdzie chodzi o wykonanie dużych obliczeń w możliwie krótkim czasie.

Obok standardowego wbudowanego do ROM Basicu coraz szerzej stosuje się na Atari nowe interpretatory tego języka, a także wygodne kompilatory pozwalające przekształcać programy napisane w Basicu w działające znacznie szybciej ich wersje skompilowane. Nie mogą one w pełni osiągnąć walorów programów napisanych w JM, stanowią jednak dogodny pomost między obu językami.

Należy podkreślić, że już w standardowym Basicu, nazywanym zwykle Atari Basic, zapewniono obok USR także inne środki ułatwiające korzystanie z wstawek maszynowych. Obok powszechnie stosowanych instrukcji PEEK i POKE do środków tych zaliczyć trzeba użyteczne, a czasem niedostępną w innych wersjach Basicu intrukcje służące do operawonia ciągami znakowymi: ADR (A$), a także pomocną programującemu instrukcję pozwalającą ustalić długość ciągu: LEN (A$). Nazwy zmiennej łańcuchowej używamy tu, oczywiście, jako przykładu, stałym natomiast wyróżnikiem takich zmiennych jest kończący ich nazwę znak "$".

W jaki sposób Basic komunikuje się z pomocą USR z podprogramem w JM? Można tu dostrzec znaczną analogię ze sposobem wiązania programów z podprogramami w samym języku maszynowym, o czym była już mowa. Analogia podstawowa i najważniejsza polega na tym, że główny tor takiej komunikacji przebiega przez s t o s. Dane niezbędne do wykonanie podprogramu przekazuje USR za pośrednictwem tej ważnej struktury danych. Posłużmy się dla bliższego wyjaśnienia sprawy prostym przykładem. Powiedzmy, że chcemy w Atari Basicu zastosować niedostępną w jego podstawowej wersji operację bitowej różnicy symetrycznej EOR. W tym celu pod określonym adresem, np. $600, umieścimy podprogram, który obliczy EOR dwóch liczb i zostawi gdzieś wynik tej operacji. Niech będą to liczby 77 i 88. Wywołanie podprogramu mieć będzie następującą postać:

X=USR (1536,77,88)

1536 - to dziesiętna wartość adresu początku podprogramu, natomiast 77 i 88 - to p a r a m e t r y przekazywane do podprogramu. W tej pozornie prostej sytuacji czeka nas jednak parę niespodzianek, co skłania do dość dokładnego przyswojenia zasad przekazywania parametrów.

Najpierw USR umieszcza na stosie adres powrotu do Basicu, czyli adres, pod którym znajduje się następna instrukcja programu. Zajmuje on dwa bajty. Z kolei USR lokuje na stosie parametry. Tu jednak ważna uwaga. Są one zawsze d w u b a j t o w y m i liczbami całkowitymi bez znaku. W naszym przykładzie bardziej znaczące bajty będą miały zatem wartości 0 . USR umieszcza na stosie najpierw młodszy bajt, a potem starszy. I wreszcie cecha ostatnia tego systemu przekazywania parametrów: na samym szczycie stosu USR umieszcza w jednym bajcie l i c z b ę przekazanych parametrów. Czyni to z a w s z e, również wtedy, gdy ta liczba wynosi 0, czyli gdy nie są przekazywane żadne parametry.

Ostatnia istotna informacja dotyczy trybu przekazywania parametrów. Są one umieszczane na stosie w kolejności odwrotnej niż wymieniona w nawiasach przy funkcji, tak więc gdy je potem pobieramy ze stosu, mają one kolejność taką jak zapisana w programie ze starszymi bajtami jako pierwszymi . Nawiasem mówiąc, D. i K. Inmanowie podają w swej książce 1 mylną kolejność parametrów.

Przyjrzyjmy się, jak będzie wyglądać stos po przekazaniu wszystkich parametrów w rozważanym przykładzie:

Szczyt+0	2	Liczba parametrów
Szczyt+1	0	Starszy bajt pierwszego parametru
Szczyt+2	77	Młodszy bajt pierwszego parametru
Szczyt+3	0	Starszy bajt drugiego parametru
Szczyt+4	88	Młodszy bajt drugiego parametru
Szczyt+5	xx	Starszy bajt adresu powrotu
Szczyt+6	xx	Młodszy bajt adresu powrotu.

Znajomość struktury przekazywania parametrów jest ważnym warunkiem ich prawidłowego wykorzystania. Musimy je zdejmować

ze stosu w kolejności odwrotnej i zabrać ze stosu dokładnie tyle parametrów, ile USR na nim umieściło, oczywiście, z wyjątkiem adresu. Jeżeli tego nie uczynimy, nastąpi nieuchronne zawieszenie się programu. Wystarczy nie zabrać jednej liczby, by adres powrotu został całkowicie zniekształcony.

W punkcie 6.7 omówiony został przykład przekazywania parametrów z jednego programu maszynowego do innego. Struktura przekazywanych tam danych była dokładnie taka sama, jak omówiona przed chwilą.

7.3 Parametry i wyniki

Kontynuując nasz przykład rozpatrz y teraz strukturę podprogramu i sposób odbioru w nim parametrów. Jeżeli nasz podprogram przeznaczony jest do wykonywania EOR na parach liczb dwubajtowych, to lokując go pod adresem EORY otrzymamy następującą jego budowę:

```
EORY PLA          Usuwamy liczbę parametrów
     PLA          Starszy bajt pierwszego parametru ...
     STA P1+1     lokujemy pod adresem P1+1
     PLA          Młodszy bajt
     STA P1       pod adresem P1
     PLA          Starszy bajt drugiego parametru
     EOR P1+1     Wykonujemy EOR ze starszym bajtem P1
     STA P1+1     Zapisanie wyniku
     PLA          Młodszy bajt P2
     EOR P1       Wykonujemy EOR z młodszym bajtem P1
     STA P1       Wynik w P1
     RTS          N i e z b ę d n y  rozkaz powrotu do Basicu.
```

USR nie tworzy uniwersalnego mechanizmu wykorzystania wyników przekazywanych z podprogramu. W danym wypadku trzeba po prostu odczytać z pomocą PEEK-ów z komórek P1 i P1+1:

```
? PEEK(P)+256xPEEK(P+1)
```

W omówionym przykładzie posłużyliśmy się konkretnymi liczbami. Dogodną stroną tego mechanizmu jest to, że dane do podprogramu przekazywać możemy również w zmiennych i wyrażeniach Basicu.

Nasz podprogram będzie działać poprawnie w przypadku dowolnych liczb nie więcej niż dwubajtowych. Powiedzmy, że adresem P1 jest 16000 dec. Możemy wówczas zbudować następujący prosty program, który pozwoli wykonywać EOR na dowolnych parach liczb:

```
10 ? "BINARNE EOR"
20 ? "podaj dwie wartości"; INPUT A,BB
30 X=USR(1536,A,B) :? PEEK(16000)+256xPEEK(16001) :END
```

Adres można, oczywiście także podać jako zmienną, a wynik wykorzystać w dowolny sposób.

7.4 Gdzie umieścic podprogram?

Stosowanie podprogramów w JM wymaga rozstrzygnięcia paru istotnych kwestii:

- jak napisać podprogram?
- jak go bezpiecznie umieścić w pamięci?
- jak go zapisać w celu wielokrotnego wykorzystania?

Odpowiedź na pierwsze pytanie jest stosunkowo prosta. Podprogram, gdy tylko jest to możliwe, powinniśmy napisać posługując się jednym z dostępnych asemblerów. Oszczędzi to nam czasu i wysiłku związanego z opracowaniem i uruchomieniem - podprogramu.

Pytanie drugie jest w oczywisty sposób istotne. Gdy w pamięci komputera znajduje się program w Basicu, trzeba dla podprogramu znaleźć bezpieczne miejsce, które zapobiegłoby jego krzyżowaniu się z programem głównym. Jednym z dobrych miejsc dla podprogramów nie dłuższych niż 256 bajtów jest na Atari strona szósta, z której skorzystaliśmy w przykładzie. Gdy program w Basicu nie jest duży, bezpiecznie jest lokować podprogram w górnych adresach RAM. Jednakże największe bezpieczeństwo programowi i najłatwiejszy do niego dostęp uzyskuje się wtedy, gdy możliwe jest uczynienie go integralną częścią listingu Basicu. Dochodzimy tu zatem do pytania trzeciego i najistotniejszego.

Są trzy podstawowe sposoby utrwalenia podprogramu w języku maszynowym:

1. Zapisanie podprogramu jako odrębnej całości w kodzie maszynowym i wgrywanie go do pamięci osobno przed uruchomieniem programu w Basicu.

2. Utrwalenie podprogramu w liniach DATA i wgrywanie go pod właściwy adres z pomocą instrukcji Basicu READ i POKE.

3. Zapisanie podprogramu w postaci ciągu znakowego wgrywanego do pamięci przez program główny podobnie jak w wariancie 2.

Rozważmy plusy i minusy tych trzech sposobów korzystania z podprogramów w JM. Z pomocą pierwszego można bez wątpienia osiągnąć efektywne wykorzystanie podprogramu pod warunkiem, że umieści się go we właściwym miejscu w pamięci. Pewnym minusem tego rozwiązania jest to, że program podzielony jest na dwie odrębne części, z których każda, zwłaszcza przy korzystaniu ze stacji dyskietek, wymaga odrębnego postępowania przy wgrywaniu do pamięci.

Dwa pozostałe rozwiązania pozwalają uniknąć tych kłopotów. W obu przypadkach podprogram staje się integralną częścią programu głównego zapisanego na taśmie czy dyskietce, a odrębność metod ich działania ujawnia się dopiero w fazie wykonania.

Zapis w postaci linii DATA jest bodaj najczęściej stosowany. Pojawia się przy nim to samo pytanie, co przy wariancie pierwszym: gdzie umieścić podprogram w pamięci? Niedogodnością tego wariantu jest również to, że tak zapisany podprogram zajmuje stosunkowo dużo miejsca, bowiem w liniach DATA w Basicu każdy bajt zapisywany jest w tylu znakach, ile zajmuje jego reprezentacja w układzie dziesiętnym, czyli np. 235 zajmuje trzy bajty, a 7 - jeden. Ponadto zajmują miejsce numery linii, instrukcje DATA i przecinki między danymi.

Trzeci sposób zapisu podprogramów wymaga dokładniejszego omówienia. Polega on na tym, że tworzymy zmienną łańcuchową, np. A$, po czym w kodach jej znaków umieszczamy cały podprogram w języku maszynowym.

Rozpatrzmy to na przykładzie małego podprogramu, który będzie wykonywał binarne AND na dwóch podanych z programu liczbach 1-bajtowych U i W oraz zapisywał wynik. Wykorzystamy do tego komórkę o adresie 0, używaną przez OS Atari tylko przy

włączaniu komputera. Podprogram mieć będzie następującą postać po prawej d z i e s i ę t n e wartości kodu maszynowego :

PLA	104	Zdejmujemy liczbę argumentów
PLA	104	Zdejmujemy starszy bajt U równy 0
PLA	104	Zdejmujemy U
STA 0	133 0	Zapisujemy w komórce o adresie 0
PLA	104	Zdejmujemy starszy bajt W równy 0
PLA	104	W jest w akumulatorze
AND 0	37 0	U AND W
STA 0	133 0	Wynik pod adresem 0
RTS	96	Powrót do Basicu.

Podprogram liczy 12 bajtów. Piszemy:

```
10 DIM A$(12)
20 FOR I=1 TO 12: READ DANE: A$(I, I)= DANE: NEXT I
30 DATA 104,104,104,133,0,104,104,37,0,133,0,96
40 POKE 766,1: ? "20 A$="; CHR$ (34);A$
```

Gdy uruchomimy ten program, wydrukuje on na ekranie linię o numerze 20 przedstawiającą czyli definiującą A$ jako niezrozumiały ciąg znaków w cudzysłowie. Trzeba teraz "przejechać" kursorem po tej linii, co wprowadzi ją do naszego programu jako nową linię o numerze 20. Zamist poprzednich dopisujemy nowe linie:

```
30 INPUT U,W
40 X=USR ADR(A$),U,W)
50 ? PEEK(0)
```

Program ze starą linią 10 i dopisanymi 20-50 jest gotów. Wydrukuje na ekranie binarne AND dwóch liczb, które podamy.

Tą lub podobną metodą można przekształcić podprogram w JM w zmienną łańcuchową, która będzie odtąd stale znajdować się w programie. Wywołując z pomocą USR jej adres – a jest on w Basicu zmienny, zależy bowiem od miejsca, gdzie znajduje się dół pamięci – uruchamiamy podprogram. Tu właśnie przydaje się instrukcja ADR.

Jakie są trudności i jakie ograniczenie tej metody? Pewną

trudność sprawia to, że nie wszystkie znaki są drukowane. Łagodzimy ją stosując POKE 766, wartość niezerowa , co w Atari powoduje, że znaki sterujące tracą swą funkcję, mogą być drukowane. Po zakończeniu trzeba przywrócić zero w tej komórce, by znaki sterujące znów pełniły swą podstawową funkcję. Poważniejszą trudnością jest to, że w łańcuchu nie możemy wydrukować dwóch znaków: Returnu - CHR$(155) oraz cydzysłowu - CHR$(34). Ten ostatni zostanie wprawdzie wydrukowany, ale nasz łańcuch będzie w tym miejscu oberwany, bowiem Basic znak cydzysłowu odczyta jako koniec łańcucha. Można wprawdzie te znaki zastąpić w łańcuchu tymczasowo innymi oraz przewidzieć wprowadzenie na ich miejsce właściwych kodów, metoda traci jednak wówczas nieco na uroku. W każdym razie przed jej zastosowaniem powinniśmy zawsze sprawdzić, czy w kodzie podprogramu nie ma liczb 155 (9B hex) lub 34 (22 hex). Czasami warto się trochę potrudzić, bowiem korzyści są oczywiste: pełne bezpieczeństwo podprogramu oraz jego obecność w tekście Basicu, dzięki czemu nie musimy wprowadzać do pamięci dwóch odrębnych programów.

Kod maszynowy włączany w ten sposób do programu musi być p r z e m i e s z c z a l n y, tzn. nie zawierać skoków bezwarunkowych ani skoków do podprogramów we w ł a s n y m obrębie, bowiem ich adresy są zmienne.

Ćwiczenia

1. Opracujmy podprogram w JM, który pozwoli wykonać binarne OR na dwóch liczbach dwubajtowych, a także program w Basicu do wydrukowania wyniku.

2. Czy i jak można uzyskać przemieszczalność podprogramu, gdy w jego wnętrzu zagnieżdżone są dalsze podprogramy?

7.5 Tworzenie kodu przemieszczalnego

W poprzednim punkcie zwróciliśmy uwagę na użyteczność kodu przemieszczalnego, to znaczy niezależnego od adresów, pod którymi znajduje się w pamięci. Osiągnięcie przemieszczalności kodu maszynowego ważne jest nie tylko w przypadku podprogramów wykorzystywanych z Basicu. W wielu wypadkach użyteczne jest umieszczenie programu w najniższym dostępnym obszarze pamięci,

a jego początek jest zmienny, podobnie jak początek wszystkich programów w Basicu.

Istnieje kilka metod osiągnięcia przemieszczalności kodu. Pierwszą i najoczywistszą jest po prostu wyeliminowanie z niego wszelkich skoków JMP i JSR. Jeżeli chodzi i JMP, to rzecz jest względnie łatwa. Można ten rozkaz zastąpić skokiem warunkowym, na przykład, stosując następujący równoważnik:

Zamiast:	JMP	Stosujemy:	CLV
			BVC

Skasowanie znacznika V z pomocą CLV sprawi, że umieszczony bezpośrednio potem rozkaz BVC zawsze będzie wykonywać odgałęzienie pod wyliczony adres. Jeżeli skok jest za krótki, można go wydłużyć metodą opisaną w punkcie 6.10.

Podobny efekt przekształcenia skoku warunkowego w bezwarunkowy można osiągnąć również z pomocą par rozkazów:

CLC	SEC	LDA #0	LDA #1	LDA #80	LDA #0
BCC	BCS	BEQ	BNE	BMI	BPL

Trudniej jest wyeliminować, o czym była mowa w ćwiczeniu, rozkaz JSR. Można to osiągnąć rezygnując z podprogramów i powtarzając ich teksty we wszystkich miejscach, z których były wywoływane. Osiąga się dzięki temu nawet pewne przyspieszenie wykonania, jednak program może się znacznie wydłużyć.

W niektórych wypadkach eliminowanie JSR w taki sposób przeczyłoby wręcz zdrowemu rozsądkowi, np. wtedy, gdy podprogram jest duży i wywoływany z kilkunastu miejsc w programie. W takim wypadku trzeba stosować inne metody. Najczęściej stosowana polega na tym, że nie zmieniając struktury programu wprowadza się do niego podprogram korygujący adresy skoków JMP i JSR, zanim jeszcze program rozpocznie wykonywanie swych właściwych zadań. W tym celu tworzy się tablice adresów zawartych w operandach wszystkich takich skoków dla pewnego wyjściowego adresu początku programu. W chwili wprowadzania programu do pamięci ustalony zostaje rzeczywisty adres początku programu przemieszczalnego. Różnicę między tym ostatnim a wyjściowym adresem początku podprogram przemieszczający dodaje do wszystkich operandów odnotowanych w tablicy.

Przypuśćmy, że wyjściowy adres początku programu wynosi 3400 hex, a operandy skoków JSR znajdują się pod adresami wyższymi od niego o 6, A, 12 i 18 hex. Gdy program zostanie wprowadzony pod adres 3200 hex, wszystkie cztery odnotowane operandy adresowe skoków JSR zostaną zmniejszone o 200 hex.

Jeżeli podprogram w JM umieszczamy w Basicu w postaci łańcucha znakowego lub w liniach DATA, korektę taką w stosunku do wielkości ADR(A$) łatwo jest wykonać z Basicu. Zapewni to całkowitą przemieszczalność kodu maszynowego, z którego korzystamy. W przypadku umieszczania tego kodu w łańcuchu znakowym wymaga jednak sprawdzenia, czy któryś bajt w tym kodzie nie przybrał wartości kodu cydzysłowu lub Returnu.

7.6 Przykład podprogramu: konwersja dec na hex

Nowsze wersje Basicu umożliwiają wprowadzanie i wyprowadzanie danych liczbowych w układzie szesnastkowym. Nie jest to możliwe w Atari Basic. Proponowany poniżej podprogram wykonuje konwersję 16 -bitowej liczby dziesiętnej na szesnastkową i wyprowadza wynik na ekran. Załączony program w Basicu pokazuje, jak można wykorzystać w celu wyświetlenia w postaci liczb szesnastkowych oraz znaków zawartości dowolnego fragmentu pamięci.

Podprogram wykorzystuje fakt, że każda połowa bajtu reprezentuje jedną cyfrę hex. Trzeba zatem dla każdego bajtu wykonać hierarchicznie trzy następujące czynności:

1. Podzielić bajt na połowy.
2. Każdą połowę przekształcić z cyfry hex w bajt zawierający odpowiadający tej cyfrze kod ASCII.
3. Wydrukować znak z pomocą podprogramu OS.

Mimo niewielkich rozmiarów naszego podprogramu możemy w nim zatem zastosować aż trzypiętrowe zagnieżdżenie podprogramów. Poniżej tekst zapisu w asemblerze. Wybraliśmy dla podprogramu adres początkowy 3400 hex czyli 13312 dec.

```
3400  PLA          Zdejmuje liczbę parametrów
      PLA          Zdejmuje starszy bajt
      BEQ 3407     Jeżeli = 0, przechodzi do młodszego
      JSR 340C     Skok do podprogramu I
```

```
3407  PLA          Zdejmuje młodszy bajt
      JSR 340C     Skok do podprogramu I
      RTS          Powrót do Basicu
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
340C  PHA          Początek podpr.I. Bajt na przechowanie
      LSR          Przesunięcie lewego półbajtu o 4 bity w
      LSR          prawo
      LSR
      LSR
      JSR 341B     Skok do podprogramu II
      PLA          Odtworzenie bajtu
      AND #0F      Maska wyodrębnia prawy półbajt
      JSR 341B     Skok do podprogramu II
      RTS          Powrót do części głównej podprogramu
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
341B  CLC          Zawsze przed dodawaniem
      ADC #30      Kod ASCII cyfry 2 = 32, 3 = 33 itd.
      CMP #3A      Czy to cyfra 0-9?
      BCC 3424     Jeżeli tak, można drukować?
      ADC #6       Jeżeli nie, trzeba jeszcze dodać 6
3424  JSR F2B0     Skok do podpr.III w systemie operacyjnym
3427  RTS          Powrót do podprogramu I
```

Krótkie wyjaśnienie do ostatniej części. Kody ASCII cyfr dziesiętnych są o 30 hex wyższe od wartości tych cyfr. Ponieważ jednak w hex posługujemy się również literami A-F, trzeba uwzględnić fakt, że każdy z tych symboli ma kod ASCII o 37 hex większy niż cyfra hex, którą reprezentuje. Np. dla "A" dziesiętnie wynosi to 65-55=10 czyli odpowiednik tej cyfry. Dlaczego zatem dodajemy tylko 6? Przyjrzyjmy się uważnie temu miejscu w programie. "BCC 3424" odsiało nam sytuację, gdy bit C był skasowany. Tak więc w miejscu rozkazu "ADC #6" znacznik C jest zawsze ustawiony. ADC dodaje go do tworzonej sumy.

Zabieg taki jest na ogół nieco niebezpieczny. Być może, trafniej byłoby wprowadzić ponownie przed dodawaniem rozkaz CLC i dodać 7. Żal jednak tego bajtu i dwóch straconych cykli zegarowych ... Żal tym bardziej, że ten skromny podprogram

można wykorzystać na różne sposoby, w tym również do szybkiego wyprowadzania na ekran obrazu dużych fragmentów pamięci komputera.

Tę właśnie możliwość wykorzystuje poniżej przedstawiony program w Basicu. Wyświetla on pamięć w postaci liczb hex oraz po prawej stronie - odpowiednich znaków. To ostatnie ułatwia ustalenie, czy w pamięci nie znajduje się jakiś tekst. W programie zastosowaliśmy poznaną już metodę drukowania znaków sterujących, by w prawej części obrazu ekranowego móc je wydrukować. Ze względu na to, że kod Returnu nie daje się drukować, a zera dawałyby dużo znaków kierowego serca, oba kody zastąpiono kodem kropki w liniach 60 i 70. 13312 - to adres początku podprogramu. Oto listing Basicu:

```
10 A=13312
20 ? "Granice obszaru";: INPUT B,C: POKE 766,1: POKE 82,0: ?
30 FOR I=B TO C STEP 8:X=USR(A,I)
40 FOR J=0 TO 7: D=PEEK(I+J): ? " ";:X=USR(A,D):NEXT J: ? " ";:
50 FOR J=0 TO 7: D=PEEK(I+J)
60 IF D=0 THEN D=46: GOTO 80
70 IF D=155 THEN D=46
80 ? CHR$(D);
90 NEXT J
100 ?:NEXT I:POKE 766,0:POKE 82,0:END
```

Przedstawiony przykład ilustruje jedną z bardzo licznych możliwości wykorzystania języka maszynowego do zapewnienia większej sprawności wykonania programów napisanych w Basicu.

<u>Ćwiczenia</u>

1. Opracujmy w Basicu program, który z pomocą przedstawionego podprogramu w JM będzie podawał wartość hex liczby dziesiętnej, którą wprowadzimy z klawiatury.

x 2. Opracujmy w asemblerze podprogram, który będzie wykonywać zadania z linii 60-80 oraz jego wywołanie z Basicu.

Rozdział 8

CZY W ASEMBLERZE MOŻNA PROGRAMOWAĆ STRUKTURALNIE?

8.1 Wstępne informacje i uwagi

Pojęcie programowania strukturalnego pojawiło się po raz pierwszy w r. 1972 w tytule głośnego artykułu Dahla, Dijkstry i Hoare'a "Structured programming", poświęconego m e t o d o m programowania. Sprawa doskonalenia tych metod dotyczy asemblera i JM, jak wszystkich języków programowania. Ważna jest dla każdego, kto pragnie pisać programy.

Skąd zatem wątpliwość wyrażona w tytułowym pytaniu rozdziału i potrzeba udzielenia na nie odpowiedzi? Źródłem nieporozumień, które powstały niedługo po pojawieniu się nowej idei, a po części utrzymują się również dziś, zdaje się być uproszczone pojmowanie samej koncepcji programowania strukturalnego. Niektórzy sądzą, że tylko w językach wysokiego poziomu, w dodatku nie we wszystkich, można programować strukturalnie. Glenford J. Myers pisał w r. 1976, że w asemblerze "programowanie strukturalne jest prawie niemożliwe" oraz zwierzał się: "Gdybym był kierownikiem działu przetwarzania danych, wszedłbym do mojego ośrodka obliczeniowego i fizycznie «wyłączył» wszystkie asemblery" [11, s. 128 i 137].

Czym jest programowanie strukturalne? Mówiąc najogólniej jest to metoda tworzenia programów o poprawnej, porządnej s t r u k t u r z e stąd nazwa . Zwolennicy koncepcji poszli dalej: sformułowali z a s a d y dobrego programowania i budowy programów.

Nie wszystkie proponowane przez nich reguły wytrzymały w pełni próbę czasu. Dotyczy to zwłaszcza sformułowanego pierwotnie wręcz zakazu używania instrukcji GOTO. Główny autor koncepcji, wybitny informatyk holenderski Edsger W. Dijkstra, trafnie dostrzegając, że nadużywanie GOTO prowadzi do pows-

tawania nieprzejrzystych programów, proponował całkowicie usunąć ową instrukcję. Późniejsze dyskusje doprowadziły do złagodzenia wymogu, wykazano bowiem, że w pewnych wypadkach GOTO jest bardziej zrozumiałe niż odpowiednie "strukturalne" formy.

Główne zasady programowania strukturalnego, tak jak się je dzisiaj na ogół pojmuje, można streścić w następujących punktach:

- program powinien składać się z niedużych jednostek zwanych modułami, w których obrębie nie ma już mniejszych podprogramów;
- w kodzie modułu powinno być tylko jedno wejście i jedno wyjście;
- kod powinien być budowany z pomocą pięciu następujących konstrukcji podstawowych: ciągu sekwencyjnego, rozgałęzienia warunkowego IF ... THEN ... ELSE, pętli powodującej powtarzanie ciągu sekwencyjnego ze sprawdzeniem warunku powtarzania przed wejściem w ciąg lub po jego zakończeniu (WHILE ... DO i REPEAT ... UNTIL) oraz instrukcji wyboru CASE;
- możliwe jest zagnieżdżenie jednych konstrukcji we wnętrzu innych;
- należy ograniczyć do niezbędnego minimum stosowanie skoku bezwarunkowego GOTO;
- program powinien zapewniać proste i jasne rozwiązanie problemu, być napisany w poprawnym stylu, przejrzysty i czytelny.

Na rysunku 8.1 przedstawiono w postaci sieci działań pięć podstawowych konstrukcji programowania strukturalnego.

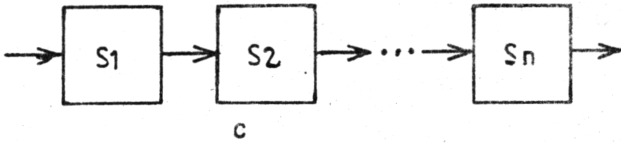
Opiszmy pokrótce ich działanie. W - oznacza warunek, a S, S1, S2 itd. - sekwencję.

a. ciąg sekwencyjny

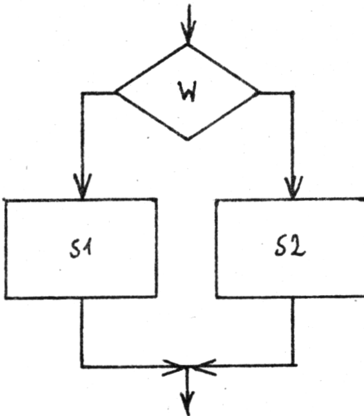
kolejne wykonywanie fragmentów programu.

b. IF W THEN S1 ELSE S2

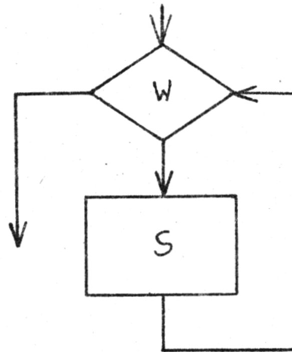
Jeżeli warunek jest prawdziwy wykonywana jest sekwencja S1, w przeciwnym wypadku S2, po czym następuje dalszy ciąg programu.



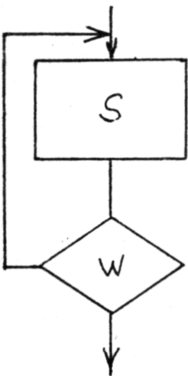
C



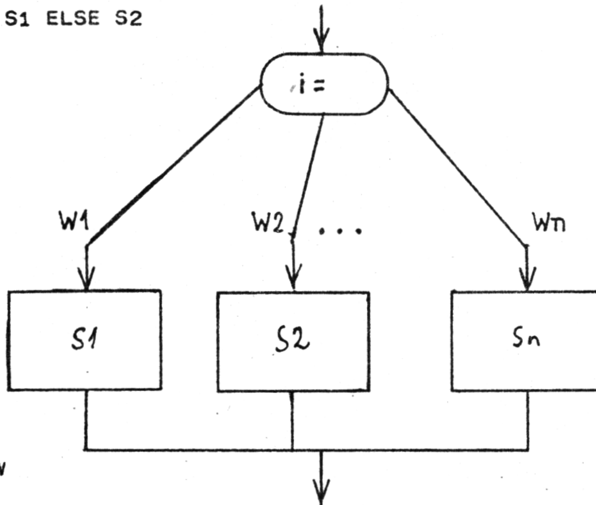
IF W THEN S1 ELSE S2



WHILE W DO S



REPEAT S UNTIL W



CASE W OF W1:S1; W2:S2; ...; WnSn

Rys. 8.1 Konstrukcje programowania strukturalnego

c. WHILE W DO S

Wykonanie sekwencji S powtarza się, dopóki warunek jest prawdziwy. Warunek sprawdzany jest przed rozpoczęciem sekwencji, toteż może ona nie być wykonana ani razu.

d. REPEAT S UNTIL W

Sekwencja powtarzana jest do czasu, aż wreszcie warunek stanie się prawdziwy, tzn. dopóki jest fałszywy. Sprawdzanie W odbywa się po zakończeniu sekwencji, toteż zawsze jest ona wykonywana co najmniej raz.

e. CASE K OF

1: S1;

2: S2;

3: S3;

. . . . .

n: Sn

Instrukcja wyboru powoduje wykonanie i-tej sekwencji, gdy sterująca jej działaniem zmienna całkowitoliczbowa K przybierze wartość i.

Na marginesie tego zestawu konstrukcji nasuwa się uwaga, że są one użyteczną pomocą w uzmysłowieniu zasad poprawnego programowania, nie powinny być jednak traktowane jako sztywne i niepodważalne schematy. Niklaus Wirth w jednym z niedawnych artykułów zauważa, że do opisu dowolnego algorytmu w zasadzie wystarczą trzy formy sterowania tokiem jego wykonania; proces sekwencyjny, konstrukcja warunkowa IF ... THEN oraz pętla WHILE ... DO. Maurer [3] wykazuje z kolei, że każdą konstrukcję programowania strukturalnego można wykonać stosując jedynie GOTO, czego zresztą nie doradza.

Konstrukcje programowania strukturalnego zostały wprowadzone w postaci instrukcji do wielu języków, w tym Pascala, Moduli, Fortha i języka C, a także do nowszych wersji Basicu. W starszych, np. w Atari Basicu, dostępne jest rozgałęzienie warunkowe w ograniczonej postaci IF ... THEN, instrukcja wyboru nazwana ON ... GOSUB, a także pętla FOR ... NEXT, sterowana przez odliczanie ze sprawdzeniem warunku na końcu sekwencji. Inne pętle trzeba realizować z pomocą GOTO.

W asemblerze nazwy konstrukcji nie występują (poza pseudoinstrukcjami kompilowania warunkowego, które mają inne znaczenie). Czy miałoby to oznaczać, że postulaty programowania strukturalnego nie mogą być zrealizowane w asemblerze? Wystarczy przejrzeć wymienione wcześniej zasady, by stwierdzić, że w s z y s t k i e mogą być efektywnie stosowane przez programującego w asemblerze i JM.

W pełni uzasadnione są postulaty dzielenia programu na mniejsze jednostki oraz zagnieżdżania jednych konstrukcji w innych. Nic nie stoi na przeszkodzie, by w asemblerze szeroko wykorzystywać podprogramy oraz rozkazy JSR i RTS zapewniające automatyczną komunikację z nimi.

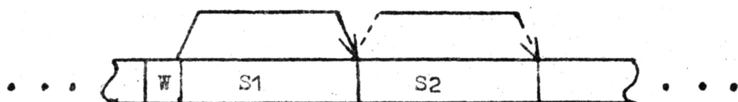
Dzięki lakoniczności swej listy rozkazów 6502 narzuca ład w programowaniu i konieczność dokładnego przemyślenia struktury algorytmu jeszcze przed napisaniem programu. Pojęcie w a r u n k u znajduje mocne oparcie w zestawie efektywnych rozkazów porównań, a przede wszystkim w tym, że znaczniki rejestru stanu procesora dostarczają pod tym względem bardzo bogatej informacji i ułatwiają sprawdzenie wszelkich warunków, z jakimi się programujący w języku wysokiego poziomu styka.

I wreszcie, bogaty repertuar rozkazów odgałęzień 6502 otwiera przed programującym szerokie możliwości tworzenia konstrukcji najbardziej efektywnych i najlepiej dostosowanych do zadania, które zamierza zrealizować.

8.2 IF ... THEN ... ELSE

Przyjrzyjmy się sieci działań tej konstrukcji na rys. 8.1. Zwraca uwagę, że w przeciwieństwie do ciągu oraz obu rodzajów pętli sprawdzenie warunku przez IF powoduje wybór jednego z dwóch r ó w n o l e g ł y c h ciągów rozkazów. Na dwuwymiarowym rysunku łatwo jest to przedstawić. Jednakże pamięć komputera bardziej przypomina jednowymiarowy odcinek prostej, oś liczbową z oznaczonymi na niej kolejnymi adresami od 0 do 65535.

Nie ma zatem innego sposobu rozmieszczenie w pamięci dwóch alternatywnych ciągów, aniżeli ich szeregowe ustawienie, w



Rys. 8.2 IF ... THEN ... ELSE w pamięci

najprostszym wypadku jednego bezpośrednio za drugim. Rysunek 8.2 schematycznie to przedstawia. Co stanie się, gdy warunek jest prawdziwy? Program wykona sekwencję 1. Gdy dojdzie do jej końca, powstanie pytanie, co robić dalej. "Na drodze" jest sekwencja 2, ale ma być ona przecież wykonana tylko wtedy, gdy warunek jest fałszywy. Trzeba wykonać s k o k do dalszej części programu. Osiąga się to przeważnie z pomocą odpowiednio dobranych rozkazów odgałęzień. Czasem niezbędny jest JMP. Z kolei, gdy warunek jest fałszywy, nie może być wykonana sekwencja 1 i musi nastąpić s k o k do początku sekwencji 2. Te dwa warianty zaznaczone zostały na rysunku z pomocą odmiennych strzałek.

Rozpatrzmy przykład programu, który będzie obliczał wartość bezwzględną liczby 8-bitowej ze znakiem. W Basicu zapiszemy to:

```
IF N>0 THEN A=N ELSE A=-N
```

W asemblerze można to rozwiązać, jak poniżej. Przyjmujemy, że przetworzona liczba zapisana będzie na powrót do komórki, z której została pobrana.

```
      LDA P       A=P
      BPL DALEJ   IF P>0 THEN GOTO DALEJ
      EOR #FF     ELSE: Odwrócenie bitów liczby
      STA P       Zapisanie nowej wartości w P
      INC P       Zwiększenie o 1. Powstała odpowiednia liczba
                  dodatnia
DALEJ  następny rozkaz
```

W tym wypadku nie trzeba było stosować skoku bezwarunkowego.

A może zadanie skomplikowałoby się, gdybyśmy przyjęli, że wartość bezwzględna liczby musi znaleźć się w akumulatorze?

```
      LDA P
      BPL DALEJ
      EOR #FF
      CLC
      ADC #1
DALEJ  następny rozkaz
```

JMP znów okazał się niepotrzebny.

Przykłady te wskazują, że z pomocą dokładnego programowania można w wielu wypadkach budować konstrukcję IF ... THEN .. ... ELSE bez konieczności uciekania się do skoków bezwarunkowych i nic nie tracąc na przejrzystości programu. Przy bardziej złożonych i większych programach może się to jednak okazać trudne lub niecelowe.

Ćwiczenia

x 1. Napiszmy w asemblerze ciąg rozkazów realizujących następujący program:

IF WAR1 THEN SEKW1 ELSE WHILE WAR2 DO SEKW2

2. Jak napisać w Basicu poniższe sekwencje instrukcji posługując się wyłącznie IF ... THEN oraz GOTO?

x a/ IF WAR THEN REPEAT Q UNTIL D ELSE SEKW

b/ WHILE WAR1 DO IF WAR2 THEN SEKW1 ELSE SEKW2.

8.3 Pętle

Oba rodzaje pętli: WHILE ... DO i REPEAT ... UNTIL, nie wywołują komplikacji, o których mowa była w poprzednim punkcie. Sekwencja powtarzanych rozkazów stanowi z reguły jedną całość, skomplikowaną ewentualnie skokami do podprogramów bądź rozrzuconą w różnych miejscach pamięci, ale powiązaną skokami. Najważniejsze pytania przy tego rodzaju pętlach brzmią: Jaką postać nadać warunkowi kontynuowania bądź opuszczenia pętli? Jak sprawdzić prawdziwość lub fałszywość warunku? Gdzie - na początku czy na końcu powtarzanej sekwencji - umieścić sprawdzenie warunku?

W pętlach stosuje się warunki dwóch przede wszystkim rodzajów. Pierwszy służy do zapewnienia, by pętla powtórzona została określoną liczbę razy. Mówimy wówczas o pętli sterowanej przez odliczanie, odliczanej (ang. counting loop) bądź sterowanej licznikiem. Pętle tego rodzaju mają odrębne instrukcje w wielu językach, w tym w Basicu.

W "klasycznych" konstrukcjach programowania strukturalnego nie wyróżnia się takiej pętli, jednak w praktyce jest ona niezbędna. Sterowanie pętli licznikiem stosuje się w JM przy zadaniach tak podstawowych, jak mnożenie i dzielenie, przemieszczanie bloków pamięci i wiele innych.

6502 rozporządza mechanizmami pozwalającymi na bardzo łatwe programowanie pętli liczonych. Rejestry indeksowe X i Y dzięki możliwości ich zwiększania i zmniejszania o 1 mogą stanowić szybko działające liczniki pętli. Ich wykorzystanie w trybach indeksowanych, a zwłaszcza adresowaniu pośrednim indeksowanym Y, zapewnia sprawny dostęp do ciągów komórek pamięci. Dzięki stosowaniu malejących wartości licznika pętli bądź omówionego w punkcie 8.6 indeksowania ujemnego można niemal całkowicie wyeliminować konieczność sprawdzania warunku zakończenia pętli z pomocą CPX lub CPY uzyskując dalsze usprawnienie jej realizacji.

Drugi rodzaj sterowania pętlą polega na sprawdzaniu innych niż licznik warunków o bardzo różnym charakterze. Pozwala to programować pętle o z góry nie określonej liczbie powtórzeń.

W przypadku obu rodzajów pętli programujący w asemblerze rozporządza dużą liczbą zróżnicowanych form s p r a w d z a n i a warunków. Podobnie jak w innych językach programowania dostępne są wszystkie typy opierania decyzji na porównaniu wartości i ustaleniu ich wzajemnych relacji takich jak: większa, większa lub równa, równa, mniejsza lub równa, mniejsza, a także cech wartości; takich jak: dodatnia, równa zeru, ujemna.

Szczególnie skutecznym narzędziem sprawdzania warunków są znaczniki rejestru P oraz ich powiązanie z rozkazami odgałęzień. Trudno jest znaleźć w językach programowania wyso-

kiego poziomu instrumenty, które zapewniałyby możliwość równie skutecznego sprawdzania bardzo zróżnicowanych sytuacji, jakie mogą pojawiać się w toku wykonywania programu. Opanowanie sztuki umiejętnego posługiwania się znacznikami i sprawdzania ich stanu - a nie jest to wbrew pozorom zadanie nadmiernie trudne - pozwala często na tworzenie rozwiązań zgoła nieoczekiwanych, oryginalnych i wysoce skutecznych.

Metody sprawdzania warunków odnoszą się, rzecz prosta, zarówno do pętli, jak i konstrukcji o postaci IF ... THEN ... ELSE.

Ostatnie z wymagających rozważenia pytań dotyczy umiejscowienia punktu sprawdzenia warunku pętli; na jej początku czy na końcu? Zauważmy, że w asemblerze i JM nie jest nazbyt istotne, czy na początku pętli WHILE ... DO warunek jest prawdziwy, a na końcu REPEAT ... UNTIL - fałszywy. Istnienie par rozkazów odgałęzień przeciwstawnie reagujących na tę samą wartość znacznika zapewnia tu znaczną swobodę wyboru. Przyjrzyjmy się sekwencji:

```
     LDA #8
CYKL LDA P,X
     STA Q,X
     DEX
     BNE CYKL
```

W tym wypadku sprawdzenie warunku wykonywane jest na końcu, ale powrót do początku pętli następuje wtedy, gdy warunek że X nie równa się zeru jest p r a w d z i w y.

Ogólnie biorąc, w asemblerze najłatwiej jest sprawdzać prawdziwość warunku kontynuowania pętli na końcu, ponieważ z pomocą rozkazu odgałęzienia osiąga się w sposób prosty powrót do początku sekwencji.

Sprawdzenia warunku na początku jest w pełni możliwe, ale nieco bardziej skomplikowane. Niech zilustruje to przykład programu wykonującego te same czynności co poprzedni:

```
     LDX #9
CYKL DEX
     BEQ DALEJ
```

```
        LDA P,X
        STA Q,X
        JMP CYKL
DALEJ   następny rozkaz
```

Wyjście z pętli nastąpi w tym wypadku na jej początku, ale wymaga to dodatkowego rozkazu skoku zapewniającego powrót na początek, gdy BEQ nie spowoduje opuszczenia pętli. W niektórych sytuacjach sprawdzanie warunku na początku pętli staje się konieczne, w innym przypadku wybralibyśmy niewątpliwie pierwszy wariant. Zauważmy, że w drugim wariancie p r a w d z i w o ś ć warunku powoduje wyjście z pętli, a więc znów jest odwrotnie niż w klasycznym WHILE ... DO.

I jeszcze jeden istotny szczegół: znajdujący się poprzednio w tym samym miejscu rozkaz BNE CYKL zastąpiliśmy przez JMP CYKL. Poprzedni rozkaz miałby tu zupełnie inne działanie. Ponieważ STA nie wpływa na znaczniki, stan znacznika Z zależałby od rozkazu LDA P,X czyli od sprawdzenia, jaką wartość została pobrana do akumulatora spod adresu Q+X. Gdyby było to zero, pętla zakończyłaby działanie. Oto jakie niespodzianki mogą nas czekać przy niedokładnym posługiwaniu się rozkazami odgałęzień.

Na zakończenie tego punktu rozpatrzmy przykład programu, w którym wewnątrz pętli znajduje się IF ... THEN.

Program oblicza 16-bitową sumę N elementów tablicy zaczynającej się pod adresem TABL zapisanym na stronie zerowej. Zakłada się, że w pierwszym zerowym bajcie tablicy podana jest jej długość, czyli N.

```
        LDA #0          Inicjalizuje zerem oba bajty sumy i Y
        STA SUMA        Młodszy bajt sumy
        STA SUMA+1      Starszy bajt sumy
        TAY             Y=0
        LDA (TABL),Y    Ładuje N
        TAY             N staje się licznikiem pętli
        CLC             Przed pierwszym dodawaniem
CYKL    LDA (TABL),Y    Pobranie elementu tablicy
        ADC SUMA        Dodanie go do sumy ...
        STA SUMA        i zapisanie pod adresem SUMA
```

```
      BCC BEZC     Skok, jeżeli C=0
      INC SUMA+1   Zwiększenie o 1 MSB sumy
      CLC          Przeniesienie wykorzystane, C trzeba skasować
BEZC  DEY          Następny element tablicy
      BNE CYKL     Wznowienie, gdy Y nie równa się zeru
      RTS
```

Ćwiczenia

1. Zmieńmy ostatni program tak, by obliczał sumę tablicy: a/ 24-bitową b/ 32-bitową

2. Który rozkaz spowoduje, że BCC BEZC wykona skok warunkowy i kiedy to się stanie?

x 3. Dlaczego rozkaz LDA (TABL),Y dwukrotnie umieszczony w tym programie ma za każdym razem inny skutek i jaki?

8.4 Konstrukcja wyboru CASE OF

Rola CASE polega przede wszystkim na tym, by odpowiednio do wartości zmiennej sterującej powodować przeskoki do rozmaitych podprogramów. W podstawowej wersji, a także w implementacji zastosowanej w Basicu, zmienna sterująca przybiera wartości od 1 do N. Można ją implementować w takiej postaci w asemblerze.

Rozszerzając nieco funkcję CASE można uzależnić wybór podprogramu od zbioru innych wartości całkowitoliczbowych, np. program BUG/65 umożliwia wybór odpowiedniej czynności przez naciśnięcie jednego z klawiszy literowych wykorzystując przy tym niemal cały alfabet.

Jednym ze sposobów realizacji CASE jest porównywanie zmie zmiennej sterującej umieszczonej w akumulatorze z kolejnymi liczbami:

```
CMP #1
BEQ JEDEN
CMP #2
BEQ DWA
. . . .
CMP #N
BEQ EN
```

Odpowiednio do zawartości akumulatora program wykona skoki pod adresy wskazane przez etykiety JEDEN, DWA, ... EN..

Inne rozwiązanie polega na umieszczeniu dwubajtowych adresów skoków w tablicy i ich wykorzystaniu w programie. Niech taka tablica znajduje się pod adresem 4000 hex i zawiera trzy adresy skoków odpowiadających kolejnym wartościom zmiennej CASE: 1, 2 i 3.

Komórki tablicy	Adres skoku bajty odwrócone
4000-4001	20 31
4002-4003	12 34
4004-4006	10 30

Przyjmijmy, że wartość zmiennej sterującej znajduje się w akumulatorze i wynosi 2. Odpowiedni adres skoku do podprogramu wynosi zatem 3412. Aby dotrzeć w tablicy do jej pierwszego bajtu, trzeba od naszej zmiennej odjąć 1, pomnożyć różnicę przez dwa i wynik dodać do adresu początku tablicy. W tym wypadku będzie to (2-1)x2+4000. Oto fragment programu, który zapewni wykonanie właściwego skoku z zastosowaniem modyfikacji adresu:

```
     ASL         A=2xA
     TAX         Przenosimy do rejestru X
     DEX
     DEX         Odejmujemy 2  w naszym przykładzie 2x2-2
     LDA 4000,X  Pobieraliśmy LSB właściwego adresu  tu 12
     STA TAB+1   LSB modyfikowanego operandu JSR
     LDA 4001,X  Pobraliśmy MSB adresu  tu 34
     STA TAB+2   MSB modyfikowanego operandu JSR
TAB  LDA FFFF    FFFF będzie zmodyfikowane na 3412
```

Efektem tej sekwencji będzie czynność wykonywana przez CASE. Raz jeszcze poznaliśmy przy tym zalety modyfikacji adresów. Oczywiście, ostatni rozkaz LDA może być zastąpiony przez inny lub przez sekwencję rozkazów zależnie od celu, w jakim tworzymy tablicę.

8.5 Organizacja programowania

Stosowanie zasad programowania strukturalnego ułatwia poprawne konstruowanie programów w asemblerze. Przyczynia się tym samym do zwiększenia efektywności programowania.

Innym ważnym środkiem, który temu sprzyja, jest poprawa organizacji pracy nad p r o j e k t e m programu. Możliwe są tu dwa podstawowe podejścia.

Pierwsze polega na tym, że najpierw w oparciu o wcześniej opracowany algorytm określamy najogólniej zarys programu i jego główne części składowe, potem zaś stopniowo dzielimy je na części coraz mniejsze aż do osiągnięcia poziomu modułów, czyli fragmentów nie zawierających już w swym wnętrzu innych modułów. Taki sposób stopniowego uszczegóławiania programu nosi nazwę projektowania zstępującego lub od góry do dołu (ang. top-down).

Drugi sposób tworzenia programu jest jakby odwróceniem poprzedniego. Zaczynamy od projektowania najprostszych podprogramów, modułów- potem łączymy je w coraz większe całości aż do otrzymania programu rozwiązującego zadanie. Metoda ta nosi nazwę wstępującej lub projektowania od dołu do góry (ang. bottom-up).

Obie metody pozwalają tworzyć programy o przejrzystej strukturze, co jest istotnym warunkiem zmniejszenia skali kłopotów przy ich uruchamianiu. W praktyce łączymy te metody. Przy opracowywaniu nowego algorytmu trafniej będzie stosować przede wszystkim pierwszą, natomiast wtedy, gdy z gotowych podprogramów pragnie się zbudować większą całość, najbardziej pomocna jest metoda druga. Nawiasem mówiąc, może być ona często przydatna w asemblerze, w którym powstały biblioteki gotowych podprogramów ułatwiające rozwiązywanie cząstkowych zadań.

W przypadku asemblera w pełni możliwe jest osiągnięcie sytuacji, gdy najogólniejszy plan przerodzi się w główny program, który składać się będzie wyłącznie z wywoływania kolejnych podprogramów, te wywołają następne itd. Proponuję pod tym kątem spojrzeć na przykładowy podprogram przedstawiony w punkcie 7.5.

Jedną z często stosowanych form programu głównego jest organizowanie go w postaci pętli bez końca. Wykonuje ona ustawicznie sprawdzanie sygnałów od użytkownika i stosownie do nich wywołuje odpowiednie podprogramy. Taką pętlę tworzą, na przykład, z reguły interpretatory języków wysokiego poziomu odczytujące dane i instrukcje, które wprowadza użytkownik. Pętlę główną stosuje się również w wielu programach użytkowych, np. w edytorach, asemblerach i programach uruchamiających. Jest ona podstawowym narzędziem sterowania grami komputerowymi.

Już na początku książki podkreślona została użyteczność graficznego przedstawiania opracowywanych algorytmów w postaci sieci działań. Metoda ta pomaga w przyspieszeniu pracy, gdy okazuje się, że program nie działa i trzeba znaleźć tego przyczyny. Rzut oka na przedstawioną graficznie konstrukcję ułatwia często wykrycie błędu.

Jedną z metod powszechnie doradzanych jako skuteczne jest ręczne sprawdzanie programu (ang. desk checking). Polega ono na tym, że nie przy komputerze, lecz przy stole sprawdzamy krok po kroku skutki każdego rozkazu, stan rejestrów i znaczników po jego wykonaniu, miejsca powrotów przy pętlach, wartości w komórkach wykorzystywanych przez program itd. Często łatwiej jest wówczas usunąć błąd niż metodą długotrwałych prób przy komputerze.

Wysoce pomocne w pracy są programy uruchamiające czyli debuggery. Powiemy o nich nieco szerzej w rozdziale 10.

Na temat zasad efektywnego i poprawnego programowania istnieje rozległa popularna i specjalistyczna literatura. Niektóre jej pozycje wymieniono w bibliografii. Spośród książek niedawno wydanych można doradzić przede wszystkim pracę Niklausa Wirtha "Wstęp do programowania systematycznego" [14].

Rozdział 9

CO POTRAFI ATARI?

9.1 Mikroprocesor a konkretny komputer

To, co zostało dotychczas powiedziane na temat programowania w asemblerze i języku maszynowym 6502, odnosi się w większości do w s z y s t k i c h mikrokomputerów wyposażonych w ten procesor. Są to m.in., przypomnijmy, kolejne wersje 8-bitowych komputerów Commodore, Apple, Acorn, Laser i innych. W przypadkach, gdy w przykładach programów odwoływaliśmy się do właściwości Atari, było to zaznaczone i użytkownik innego komputera mógł zazwyczaj bez trudności dokonać ewentualnej korekty.

Co jest wspólne, a co odrębne w komputerach wyposażonych w ten sam mikroprocesor 6502? Wspólne są: lista rozkazów asemblera i JM oraz tryby adresowania, a zatem p o d s t a w o w y j ę z y k, w jakim przekazuje się komputerowi zadania do wykonania.

Różnice pojawiają się, gdy wykorzystuje się możliwości systemu operacyjnego komputera i szczególne cechy jego architektury. W każdym komputerze mikroprocesor 6502 wykonuje tak samo operacje arytmetyczne na liczbach całkowitych. Gdy jednak chcemy przejść na arytmetykę zmiennopozycyjną, natrafiamy na istotne różnice wynikające przede wszystkim z odmiennej reprezentacji liczb rzeczywistych. W każdym komputerze 6502 przetwarza teksty posługując się kodami znaków. Cóż jednak poradzimy na to, że co komputer, to inna wersja kodu ASCII?

Najistotniejsze różnice dotyczą dwóch dziedzin: przerwań i obsługi wejścia-wyjścia. W przerwaniach wspólne są podstawowe rozkazy 6502 (SEI, CLI, RTI), kategorie przerwań i komórki adresowe z nimi związane (FFFA-FFFF hex), natomiast systemy wykorzystania przerwań są rozmaite. W drugiej z wspomnianych

dziedzin 6502 nie ma odrębnych rozkazów, a zatem nie tworzy żadnych ograniczeń dla różnic. Dlatego omówienie obu kwestii odłożyliśmy do tego rozdziału.

Zapewnienie wysokiej jakości programów w asemblerze i JM nie jest w istocie możliwe bez dokładnej znajomości cech konkretnego komputera. Rozdział niniejszy stanowi ogólny przegląd niektórych cech 8-bitowych komputerów Atari. Zachęcam Czytelników, by poszerzali ten zasób wiedzy w oparciu o dalsze lektury i własną praktykę programowania. Warto wskazać, że asembler jest podstawowym językiem, w którym opracowano profesjonalnie na Atari setki wysoce sprawnych programów, w tym kompilatorów i interpretatorów języków wysokiego poziomu, systemów obsługi wejścia-wyjścia, programów użytkowych, a także rozrywkowych.

9.2 Od Atari 400 do 130XE

W r. 1977 pojawiły się pierwsze mikrokomputery domowe. W dwa lata później firma Atari specjalizująca się w wytwarzaniu gier telewizyjnych wprowadziła na rynek swoje mikrokomputery: Atari 400, a wkrótce potem Atari 800. Oba oparte zostały na mikroprocesorze 6502. Różniły się klawiaturą.

Konstruktorzy Atari 400/800 zaprojektowali specjalnie dla tego komputera trzy dodatkowe układy scalone LSI do obsługi obrazu, dźwięku,przerwań itd. System operacyjny i interpretator Basicu wgrywane były z modułów (kartridżów) . Komputery z miejsca zdobyły duże powodzenie na rynku.

Jesienią 1983 r. w celu dotrzymania kroku rywalom w zaostrzającej się konkurencji firma wprowadziła na rynek nowe komputery: 16-kilobajtowy Atari 600XL i 64-kilobajtowy Atari 800 XL. Różniły się między sobą tylko wielkością RAM. Były rozwinięciem swych poprzedników. Przy utrzymaniu wszystkiego, co się w nich okazało udane, dokonano dość głębokich zmian. Nieco zmodyfikowany interpretator Basicu,w pełni kompatybilny z poprzednim, wprowadzono do ROM. Praktycznie od nowa napisano udoskonalony, powiększony system operacyjny, który znalazł się również w ROM. Istotną zmianą było także to, że ROM Basicu i prawie cały ROM OS otrzymały dublujący je RAM, którym mogą być zastąpione.

mogą być zastąpione.

Wypadki potoczyły się nieco paradoksalnie, bowiem po wytworzeniu bardzo udanego komputera firma znalazła się w obliczu bankructwa. Wykupił ją dotychczasowy właściciel Commodore Jack Tramiel. W walce konkurencyjnej posłużył się radykalnym obniżeniem cen komputerów Atari.

Ich rodzina wzbogaciła się o kolejny model: Atari 130XE. Jest on oparty w zasadzie na takich samych układach jak 800XL. Różnice polegają zwłaszcza na zwiększeniu istniejącej już w tamtym modelu dodatkowej pamięci do 64KB, a także na zastosowa waniu nowej obudowy. Zachowując taką samą jak 800XL przestrzeń adresową Atari 130XE ma łącznie 128 KB RAM.

Firma Atari włączyła się również do rywalizacji w dziedzinie produkcji komputerów o większej długości słowa, w tym przeznaczonych do zastosowań zawodowych. Z zapowiedzi firmy wynika, że nie zamierza rezygnować z kontynuowania produkcji mikrokomputerów 8-bitowych.

Atari 800XL i 130 XE zdobyły znaczną popularność w Polsce, do czego przyczyniła się ich sprzedaż przez Pewex. Szerokim strumieniem dotarło również do naszego kraju oprogramowanie.

Gorzej przedstawia się sprawa z literaturą. Najważniejsze pozycje, wymienione w bibliografii, dostępne są głównie w języku angielskim. W dodatku dotyczą przeważnie poprzedniej wersji Atari. Najsolidniejszym opracowaniem uwzględniającym nowsze wersje jest drugie wydanie "Mapping the Atari" Iana Chadwicka [7]. Uzupełnienia dokonano w nim jednak w postaci aneksów, co wymaga uważnego porównywania ich z tekstem podstawowym, który dotyczy Atari 400/800. Książka pt. "Atari Intern" [5] uwzględnia wersję XL. Natomiast godna polecenia praca zbiorowa "De Re Atari" dotyczy "starego" Atari.

Czytelnik tych książek nie zawsze może być pewien, czy informacje są aktualne. Co więcej, zamęt informacyjny przenika również do naszych czasopism, które opisując XL podają nieraz dane dotyczące poprzedniej wersji.

Omówione kłopoty mogą być uciążliwe zwłaszcza dla programującego w JM, gdyż np. posługiwanie się nieaktualnymi adresami wysyła program na manowce.

9.3 Architektura i mapa pamięci

W rozdziale 2 omówiliśmy ogólny schemat budowy mikrokomputerów. Podstawowy układ Atari jest, oczywiście, taki sam. Jednakże obok RAM, ROM i standardowego układu wejścia-wyjścia PIA (Peripheral Interface Adapter), Atari ma również podłączone do tej samej szyny trzy specyficzne układy scalone (LSI): ANTIC (Alphanumeric Television Interface Controller), GTIA (Graphics Television Interface Adapter) i POKEY (Potentiometer and Keyboard Controller). Układy te w poważnej mierze kształtują oblicze Atari. Są one znacznie bardziej skomplikowane niż 6502.

Wszystkie cztery układy scalone i mikroprocesor działają równocześnie. Przy projektowaniu komputera zminimalizowano możliwe konflikty między nimi. Jedyny na poziomie sprzętowym powstaje wtedy, gdy ANTIC potrzebuje szyn do przekazania informacji o obrazie. Na ten czas wstrzymywana jest praca 6502. W synchronizacji pracy układów dużą rolę odgrywa rozbudowany mechanizm przerwań.

Każdy z układów dostępny jest za pośrednictwem nielicznych, lecz mających duże znaczenie rejestrów i wektorów. Obok rejestrów sprzętowych w wydzielonych obszarach przestrzeni adresowej omawiane układy mają także tzw. rejestry-cienie (ang. shadow register) w RAM.

Rejestrami takimi rozporządza również system operacyjny, toteż warto wyjaśnić ich funkcje. Znaczna część rejestrów w ROM odznacza się tym, że dane wprowadzone do nich przechowywane są przez bardzo krótki czas, po czym nikną, co może powodować załamanie się programu. Dlatego właśnie system operacyjny po włączeniu komputera lub jego gorącym starcie zajmuje dla swych potrzeb duże fragmenty RAM strony zerowej i następnych, gdzie m.in. lokuje rejestry-cienie. Informacja do nich wpisana odczytywana jest przez odpowiednie układy co pięćdziesiątą część sekundy. Zapobiega to jej zagubieniu.

Omówmy wstępnie funkcje poszczególnych układów.

PIA jest wyspecjalizowanym układem scalonym stosowanym również w innych komputerach opartych na 6502. W szczególnej konfiguracji Atari rola PIA jest ograniczona głównie do obsłu-

gi wejść joysticków oraz do sterowania jednostką zarządzania pamięcią (Memory Management Unit - MMU). MMU jest odrębnym układem wprowadzonym do Atari 600/800XL w związku z zastosowaniem dublującego ROM. PIA ma dwa porty: A i B. Port A służy do obsługi wejść dwóch joysticków, natomiast port B w przeciwieństwie do dawnego Atari, w którym obsługiwał dalsze dwa joysticki, wspiera obecnie MMU. W porcie tym o adresie D301 (54017) bity sterują mechanizmem przełączania banków pamięci według następującego schematu:

b0 - ma wartość 1, gdy włączony jest ROM, a 0, gdy na to miejsce wprowadzony jest RAM
b1 - 0, gdy Basic włączony, 1, gdy wyłączony
b2-b5 - wykorzystywane w 130XE do przełączania dodatkowych banków pamięci
b6 - nie używany
b7 - kontroluje obszar pamięci 5000-57FF. Gdy 1 - włączony jest RAM, gdy 0 - test komputera (BYE z Basicu).

Port A ma rejestry-cienie dla każdego z joysticków pod adresami 278 i 279 hex. Poza tym PIA generuje dwa przerwania na szynie szeregowego wejścia-wyjścia. Nie są one wykorzystywane przez OS, ponieważ w Atari PIA nie służy do przesyłu danych.

ANTIC - to pełnowartościowy niezależny mikroprocesor pracujący równolegle z 6502 i odgrywający zasadniczą rolę w budowie obrazu ekranowego w drodze bezpośredniego dostępu do pamięci (ang. direct memory access - DMA), tzn. bez udziału CPU. ANTIC przejmuje od 6502 dużą część zadań wyłączając na ten czas mikroprocesor. Rozmiary tej pracy ilustruje fakt, że gdy wyłączymy ANTIC, a więc również obraz, 6502 wykonuje zadania o przeszło 25 proc. szybciej. Wyłączenie ANTIC-u osiąga się przez wprowadzenie 0 do rejestru sterowania DMA pod adresem 22F (559 dec). ANTIC steruje ponadto przerwaniami niemaskowalnymi NMI.

GTIA - to kolejny wyspecjalizowany układ scalony. Jego głównym zadaniem jest przekształcanie danych ANTIC-u w obraz ekranowy, przy czym GTIA określa kolor tła i obiektów na ekranie. ANTIC kontroluje większość funkcji GTIA, w tym bardzo

użyteczną nie tylko w grach grafikę graczy-pocisków ang. player-missile graphics - PMG , która umożliwia sprawną animację obrazu.

GTIA zapewnia szerokie możliwości kolorystycznego wzbogacenia obrazu ekranowego dzięki szczególnym sposobom interpretacji danych obrazu dostarczanych przez ANTIC. GTIA określa kształt trybów graficznych oznaczonych przez w Basicu numerami 9, 10 i 11 odmiennie interpretując program ANTIC-u dla trybu 8.

Jedną z funkcji GTIA jest zapewnianie informacji o tym, czy naciśnięte są klawisze START, SELECT lub OPTION oraz guzik joysticku. Tu także tworzony jest tzw. "klick" - dźwięk powstający przy naciśnięciu klawisza.

Programujący może komunikować się z GTIA za pośrednictwem 32 komórek rejestrowych, przy czym w części z nich mieszczą się p a r y rejestrów: jeden dla czytania, a drugi dla zapisu danych. Dla wielu rejestrów GTIA system operacyjny zapewnia rejestry-cienie w RAM. W szczególności kopie dziewięciu rejestrów barw, w tym czterech dla PMG, znajdują się pod adresami 2C0-2C8 (704-712 dec). Cztery pierwsze dotyczą PMG. Wartości wpisywane do tych rejestrów określają barwy rozmaitych elementów obrazu ekranowego i tła.

POKEY wykonuje niemal wszystkie funkcje związane ze sterowaniem dźwiękiem. Może on zarządzać jednocześnie czterema kanałami, a dla każdego określa wysokość, głośność oraz rodzaj ewentualnych zniekształceń dźwięku, co umożliwia symulowanie szumów, wybuchów itd. Przez sumowanie dźwięków pary kanałów można uzyskać brzmienia podobne do wytwarzanych przez fortepian, skrzypce i inne instrumenty. Możliwe jest także rozszerzenie skali wysokości generowanych dźwięków.

Ponadto POKEY spełnia ważne funkcje w sterowaniu portem szeregowego wejścia-wyjścia i układami klawiatury, generuje liczby losowe, kontroluje paddle czyli analogowe odpowiedniki joysticków. W POKEY skupione jest sterowanie przerwaniami na żądanie IRQ. W pełnieniu tych różnorodnych funkcji POKEY wykorzystuje własne zegary systemowe.

Komunikację z POKEY-em zapewnia 16 komórek rejestrowych kumulujących w większości odrębne rejestry dla zapisu i od-

czytu danych. Część z nich ma rejestry-cienie. Z działaniem POKEY-a łączy się istnienie jeszcze jednego kodu, a mianowicie kodu klawiatury wytwarzanego przy każdym naciśnięciu klawisza - pojedyńczo lub w kombinacji z klawiszami SHIFT i CONTROL. W rejestrze CH pod adresem 2FC 764 dec pojawia się wartość kodu klawiatury ostatnio naciśniętego klawisza.

Na marginesie tego omówienia podstawowych składników konfiguracji Atari warto rozważyć pytanie, z jaką szybkością pracuje w tym komputerze mikroprocesor 6502. Podaje się na ten temat rozmaite dane. Najbardziej prawdopodobna wydaje się być informacja Chadwicka 7 . Według niej europejskie Atari pracują o 25 proc. szybciej niż amerykańskie:z cyklem 2216 MHz. Ponieważ cykl zegarowy 6502 wynosi dwukrotnie mniej, wynikało by z tego, że 6502 wykonuje u nas ponad 1,1 miliona cykli na sekundę. Oczywiście, realne tempo wykonywania programów jest mniejsze, bowiem, jak wspomnieliśmy, ponad 20 proc. czasu "kradnie" ANTIC, dalszych kilka czy kilkanaście procent odpada na odświeżenie ładunków w RAM, dochodzą do tego iinne przerwania. Czy jednak mimo wszystko nie jest to dla laika szybkość oszałamiająca?

Poświęćmy chwilę uwagi strukturze ROM Atari. Składa się on z dwóch części sąsiadujących z sobą w przestrzeni adresowej. W pierwszej, 8-kilobajtowej, mieści się interpretator Basicu. Gdy nie korzysta się z Basicu, ten odcinek ROM zastępowany jest przez RAM. Gdy dołączony jest moduł (kartridż), wykorzystuje tę przestrzeń, a przy większych modułach także niżej położonych 8 kilobajtów pamięci.

Najwyższych 16 KB przestrzeni adresowej zajmuje przede wszystkim system operacyjny. Jego rzeczywisty rozmiar jest mniejszy o 2 KB, ponieważ w obszarze tym znajdują się strefy nie do wykorzystania z rozsianymi w nich rejestrami czterech wcześniej opisanych układów sca onych.

Poniższa skrócona mapa pamięci przedstawia funkcje poszczególnych odcinków pamięci operacyjnej Atari.

Adresy (hex)	Z a w a r t o ś ć
00-FF	Strona zerowa
100-1FF	Stos

200-5FF	Obszar RAM zarezerwowany dla systemu operacyjnego
600-6FF	Strona 6, na ogół dostępna dla programów w JM
700- /	Początek DOS lub wolnego RAM
1540-3306	W DOS 2.5 obszar zajęty przez DUP.SYS
3307-7FFF	Wolny obszar RAM
8000-9FFF	Wolny obszar RAM lub moduł (kartridż)
A000-BFFF	Trzy sposoby wykorzystania: a/ ROM Basicu, b/ wolny obszar RAM c/ moduł
C000-CBFF	ROM OS. Manipulatory przerwań
CC00-CFFF	ROM OS. Międzynarodowy zbiór znaków CHARSET2
D000-D7FF	Obszar rejestrów, poza nimi nie do wykorzystania.
w tym:	
D000-D0FF	GTIA do D01F. Rejestry grafiki, graczy-pocisków, przycisków joysticków, klawiszy konsoli
D100-D1FF	Nie używane
D200-D2FF	POKEY do D20F. Rejestry dźwięku, paddle, generator liczb losowych (D20A), bajt danych szeregowego wejścia-wyjścia (D20D), wektor przerwań IRQ (D20E)
D300-D3FF	PIA do D302
D400-D5FF	ANTIC do D40F. Rejestry związane z tworzeniem obrazu oraz PMG, rejestr przerwań programu ANTIC-u i synchronizacji pionowej WSYNC (D40A), rejestry przerwań niemaskowalnych NMI:NMIEN (D40E) oraz NMIRES i NMIST (D40F)
D600-D7FF	Nie używane
D800-DFFF	ROM OS. Pakiet matematyki zmiennopozycyjnej
E000-E3FF	Podstawowy zbiór znaków CHARSET1
E400-E47F	Tablica skoków do zmienionych w XL i XE adresów podprogramów OS
E480-FFED	Podprogramy OS, m.in. scentralizowanego wejścia-wyjścia CIO, szeregowego wejścia-wyjścia (SIO), ekranu (SIN), Klawiatury (KCB), różne-

	funkcje edytora, 192-bajtowa tablica przeliczania kodów klawiatury na ASCII od FB51, rejestry przerwań związanych z klawiaturą i obrazem itd.
FFEE-FFF9	Sumy kontrolne i dane identyfikacyjne komputera
FFFA-FFFF	Wektory programów obsługi przerwań. Kolejno: NMI, RESET i IRQ

W praktyce programowania w asemblerze na Atari istotnym dla systemu adresom przypisuje się etykiety będące mnemonicznymi skrótami angielskich opisów ich funkcji. Np. RAMTOP - to etykieta adresu 6A, gdzie podaje się liczbę aktualnie dostępnych stron RAM, RTCLOK - etykieta adresu zegara, 12-14 hex. Stosuje się setki takich etykiet.

9.4 Tworzenie obrazu przez ANTIC

Działanie mikroprocesora ANTIC - to obszerny i ciekawy temat, który zmuszeni jesteśmy omówić bardzo zwięźle.

Komputer tworzy obraz na ekranie telewizora lub monitora. Urządzenia te w przedstawianiu obrazu posługują się rastrem czyli gęstą siatką punktów o różnych odcieniach szarości z ewentualnym dodaniem barw. W telewizorze brzegi obrazu są z reguły schowane za ramką, W przypadku komputera prowadziłoby to do utraty niektórych informacji. By temu zapobiec, Atari tworzy na ograniczonej powierzchni obraz złożony ze 192 linii poziomych po 320 punktów w każdej.

Informacje niezbędne do tworzenia obrazu zawarte są w tzw. pamięci ekranu - ang. screen memory . W jej kolejnych bajtach mieszczą się informacje określające rozmieszczenie punktów o rozmaitym kolorze.

ANTIC rozporządza 14 trybami graficznymi. Stosuje się w nich rozmaite wymiary najmniejszego elementu obrazu zwanego pikselem ang. picture element - pixel , a także niejednakową liczbę kolorów. Im drobniejsze są piksele i im więcej kolorów tym pamięć musi być większa. W trybach wielobarwnych informacje o kolorach użytkownik umieszcza w rejestrach GTIA. Mechanizm tworzenia obrazu na podstawie danych zawartych w pamięci

ekranu skupiony jest w ANTIC-u.

ANTIC ma własną listę rozkazów odmienną niż 6502 i posługuje się własnym programem określającym rozmieszczenie obrazu na ekranie, zwanym listą obrazu (ang. display list - DL). Określenie nie brzmi najlepiej po polsku, ale dość dobrze odpowiada istocie programu. Stanowi on bowiem cykliczną, stale powtarzaną listę stanowiącą swoisty szablon wykorzystania danych zawartych w pamięci ekranu. Program ANTIC-u wskazuje:

a/ gdzie znajduje się pamięć ekranu;

b/ w jakim trybie czy trybach graficznych mają być interpretowane dane;

c/ czy i jakie dodatkowe środki(np. przerwania, płynny przesuw obrazu) mają być zastosowane.

Standardowo pamięć ekranu umieszczana jest na szczycie dostępnego w RAM, a lista obrazu DL - tuż pod nią. Jednakże ANTIC zapewnia pod tym względem dużą elestyczność. Programujący może umieścić DL w dowolnym dostępnym miejscu w RAM. Wpisanie adresu do komórek 230-231 (560-561 dec) z młodszym bajtem jako pierwszym spowoduje, że ANTIC będzie ją wykonywał. Z kolei w DL podaje się adres początku danych w pamięci ekranu. Zapewnia to swobodę wyboru miejsca ma listę obrazu, jak i na dane. Adres początku tych ostatnich można odczytać w komórkach 58-59 (88-89 dec). Tak więc sekwencja (liczby hex):

```
     LDA #21
     LDY #0
CYKL STA (58),Y
     INY
     BNE CYKL
```

spowoduje, że 256 jednakowych znaków umieszczonych zostanie w górnych wierszach ekranu. Jakie to będą znaki? Otóż wyjaśnienia wymaga, że ANTIC w wyprowadzaniu znaków posługuje się kodem wewnętrznym, a nie kodem **ATASCII.** Poznajemy zatem już trzeci kod znaków w Atari. Kod ten dla poszczególnych odcinków wartości jest przesunięty w stosunku do kodu ATASCII przez dodanie lub odjęcie stałej wielkości. Część kodów jest taka sama. Oto tablica konwersji kodu ATASCII na wewnętrzny:

Kod ATASCII	Modyfikacja	Kod wewnętrzny
00-1F	+40	40-5F
20-5F	-20	00-3F
60-7F	00	60-7F
80-9F	+40	C0-DF
A0-DF	-20	80-BF
E0-FF	00	E0-FF

Z tablicy wynika, że kody wewnętrzne znaków specjalnych, cyfr i dużych liter są o 20 hex niższe od kodów ASCII i ATASCII.

Lista rozkazów, którą posługuje się ANTIC w swym programie, jest bardzo zwięzła. W jednym bajcie mieści się niejednokrotnie więcej niż jeden rozkaz, bowiem kody wielu z nich powstają przez ustawienie pojedyńczego bitu.

Duże znaczenie ma rozkaz tworzenia pustych linii ekranowych. W celu zapobieżenia ucieczce najwyższych linii poza ekran programy ANTIC-u zaczynają się z reguły od wpisania u góry 24 pustych linii ekranowych, czyli np. w podstawowym trybie tekstowym trzech pustych wierszy tekstu. Rozkazy te, zależnie od liczby tworzonych linii mają następujące wartości:

Kod	00	10	20	30	40	50	60	70
Liczba pustych linii	1	2	3	4	5	6	7	8

Dwa dalsze rozkazy określa się skrótami LMS (load memory scan - załaduj linię z pamięci) i JMP (rozkaz skoku).

LMS wskazuje adres w pamięci ekranu, jest zatem rozkazem trzybajtowym. Jego kod operacji powstaje przez ustawienie bitu b6 czyli ma wartość 40 hex. W tym samym bajcie umieszcza się również rozkaz określający tryb graficzny linii.

JMP ANTIC-u jest również rozkazem 3-bajtowym. Powoduje skok w obrębie listy obrazu. Ma kod operacji 1 lub w połączeniu z rozkazem "skocz i czekaj na puste przerwanie pionowe" 41.

Rozkazy dotyczące opcji specjalnych powstają przez ustawienie wskazanych poniżej bitów w bajtach innych rozkazów. Cztery najniższe bity przeznaczone są na rozkaz określający tryb grafiki. Oto znaczenia bitów:

Bity	Wartość hex	Czynność
7	80	Przerwanie programu ANTIC-u
6	40	Omówiony już LMS
5	20	Wykonuj płynny przesuw pionowy obrazu
4	10	Wykonuj płynny przesuw poziomy obrazu
3-0	F-2	Tryby:
		0010-0111 bin - tryby tekstowe
		1000-1111 bin - tryby graficzne.

Wyjaśnimy pokrótce zasady budowy trybów tekstowych i graficznych określanych w Atari przez ANTIC. Każdy taki tryb określa sposób interpretowania i wykorzystywania danych zawartych w pamięci ekranu, a mianowicie: wielkość piksela, liczbę możliwych do zastosowania kolorów oraz rozłożenie tych informacji w kolejnych bajtach.

Mechanizm ten jest szczególnie prosty w trybach tekstowych. Informacja zapisywana jest w pamięci obrazu w wewnętrznych kodach znaków. Każdy znak ma szerokość i wysokość ośmiu punktów, toteż do jego przedstawienia potrzebne są 64 bity czyli 8 bajtów. Jednakże tych 8 bajtów nie umieszcza się w pamięci ekranu. Znajdują się one w zbiorze znaków w ROM i stamtąd są pobierane, tworząc razem jeden piksel czyli jeden podstawowy element trybu tekstowego. W trybie 0 Basicu standardowo są 24 linie po 40 pikseli w każdej. W trybie 2 Basicu obie wielkości są mniejsze o połowę.

Bardziej zawiła jest organizacja trybów graficznych. Tu obok informacji o wielkości piksela trzeba często w bajtach danych przewidzieć również bity na informację o kolorze. ANTIC nie tworzy koloru, lecz dostarcza GTIA informację o nim. Wszystko to decyduje o dość skomplikowanej budowie bajtów danych i o różnych wymaganiach, jeżeli chodzi o wielkość pamięci ekranu. Dane na ten temat zawiera instrukcja.

Numery trybów stosowane w rozkazach ANTIC-u są inne niż te, które w Basicu określamy instrukcją GRAPHICS. Odpowiedniość trybów charakteryzuje poniższe zestawienie. Tryb 3 ANTIC-u (litery z przedłużonymi ogonkami w polu 8x10) nie ma odpowiednika w Basicu.

ANTIC hex	2	3	4	5	6	7	8	9	A	B	C	D	E	F
BASIC dec	0	-	12	13	1	2	3	4	5	6	14	7	15	8

Z kolei tryby 9-11 Basicu oparte są na tej samej liście obrazu trybu F.

Po dotychczasowych informacjach zrozumiała będzie budowa programu ANTIC-u zwanego listą obrazu. Przedstawimy ją dla trybów ANTIC-u 2 i 7 czyli Basicu 0 i 2 z oknem. Rozkazy powtarzające się wydrukowano dla oszczędności miejsca w ciągu.

```
70  70  70     24 puste linie ekranowe u góry
42             LMS i w tym samym bajcie tryb pierwszej linii 2
20
7C             Dwubajtowy adres początku pamięci obrazu.
               Teraz następują rozkazy druku pozostałych 23 li-
               nii w trybie graficznym 0  2 ANTIC-u :
02  02  02  02  02  02  02  02  02  02  02  02  02  02  02
02  02  02  02  02  02  02  02
41              Skocz i czekaj na puste przerwanie pionowe
E0
7B              Adres skoku do początku listy obrazu 7BE0
```

Tryb 7

```
70  70  70     24 puste linie ekranowe
47             LMS i tryb 7 ANTIC-u
70
9E             Adres początku pamięci obrazu 9E70
               Teraz 9 pozostałych linii trybu 7:
07  07  07  07  07  07  07  07  07
42             Nowy rozkaz LMS dla okna w trybie 2
60
9F             Nowy początek pamięci obrazu dla okna tekstowego
02
02
02             Trzy pozostałe linie okna tekstowego w trybie 2
41             Skocz i czekaj na puste przerwanie
```

58
9E Adres skoku do początku listy obrazu 9E58.

Pierwsza lista obrazu liczy 32 bajty, druga jeszcze mniej.

Wielorakie są możliwości korzystania w programach na 6502 z faktu, że równolegle z nim działa drugi mikroprocesor. Można, na przykład, zbudować listę obrazu, w której poszczególne linie będą wyświetlane w rozmaitych trybach, łączyć teksty z grafiką itd. Ważne jest tylko, by dane odpowiadały zaprojektowanemu układowi. Łącznie, przy 24 pustych liniach na początku, reszta powinna dawać w sumie 192 pełne linie ekranowe. Do jednej listy obrazu można zbudowac kilka zestawów danych i przełączać je przez zmianę adresów rozkazów LMS.

ANTIC tworzy także wiele innych możliwości, których w szczegółach nie sposób tu omówić. Możliwe jest zapewnienie płynnego przewijania czyli przesuwania obrazu w kierunku pionowym i poziomym. Można również tak budować dane obrazu, by obejmowały obszar większy niż ekran. Ten ostatni staje się wówczas jakby oknem, przez które oglądamy znacznie większe całości. Przykładów takich dostarczają m.in. znane programy użytkowe Synfile+ i Visicalc, a także liczne gry. Dodatkowe możliwości wzgobacenia obrazu powstają przy wykorzystaniu dwóch przerwań ANTIC-u, o których powiemy w punkcie 9.7.

9.5 Grafika graczy-pocisków

Jedną z właściwości Atari, która znacznie upraszcza programowanie animacji obrazu i umożliwia osiągnięcie dużej szybkości ruchu obiektów na ekranie jest grafika graczy-pocisków. Ułatwienia w animacji, jakie zapewnia PMG, mogą być użyte także do poważnych zastosowań. W bardzo udany sposób wykorzystano PMG w interpretatorze Logo, dzięki czemu dzieci i młodzież, korzystająca z tego języka mogą posługiwać się aż czterema żółwiami w tworzeniu grafiki, która jest ważną domeną Logo.

Jeżeli z pomocą kodu maszynowego spróbujemy uzyskać dostęp do pojedyńczego bajtu ekranowego o współrzędnych X i Y, to okaże się, że potrzeba na to ponad 30 rozkazów. PMG umoż-

liwia uzyskanie znacznie szybszego ruchu na ekranie z pomocą krótkich i prostych programów.

Grafika graczy-pocisków zakłada definiowanie obiektów o dowolnej wysokości i szerokości 8 bitów. Ograniczenie szerokości złagodzić można łącząc paru graczy w jeden obiekt, stosując szybką zmianę obrazu i innymi metodami. PMG rozporządza ponadto możliwością zdefiniowania, czy gracz lub pocisk ma przesuwać się za obiektami tła, czy przed nimi. Zapewnia kontrolę kolizji czyli ekranowych "zderzeń" pocisk-gracz, pocisk-pole gry, gracz-gracz i gracz-pole gry. Możliwości te szeroko wykorzystuje m.in. interpretator Logo na Atari.

9.6 System operacyjny

Głównym zadaniem systemu operacyjnego Atari jest zapewnienie wykorzystania wszystkich zasobów komputera, na które składają się: 6502, RAM,i ROM i cztery wspomniane układy LSI, a także zapewnienie współpracy komputera z licznymi urządzeniami zewnętrznymi: telewizorem lub monitorem, klawiaturą, joystickami, paddle, magnetofonem, stacją lub stacjami dyskietek, drukarką, modemem.

System operacyjny Atari 800XL został, jak powiedzieliśmy, gruntownie zmieniony i znacznie rozbudowany w stosunku do OS Atari 400/800. Zmianie uległy przy tym adresy wielu podprogramów i rejestrów w ROM. Jest to główną przyczyną, że część programów na dawne Atari "nie chodzi" na nowym. Często stosowanym sposobem zaradzenia kłopotom jest zastępowanie RAM-em obszaru OS i wpisywanie na to miejsce dawnego OS Atari 400/800. Wykonuje to wiele programów, m.in. Translator.

Walory nowego systemu operacyjnego są bezsporne. Jest on logicznie przemyślanym i starannie uporządkowanym zbiorem wielkiej liczby mniejszych i większych podprogramów, do których możemy się odwoływać w naszych programach.

Jak to wykonać? OS rozporządza tysiącami adresów we własnym obszarze, a także na zajętych do jego potrzeb częściach stron 0-5.

Adresy rejestrów-cieni w Atari 800 XL nie uległy zmianie w stosunku do wcześniejszych wersji. Rejestry te pełnią różne

funkcje. Czasem dla uzyskania pożądanego efektu trzeba do nich wpisać d a n e. Np. do rejestrów-cieni możemy wpisywać dane określające kolory poszczególnych elementów obrazu ekranowego. Inne rejestry są słowami dwubajtowymi, które zawierają a d r e s. Noszą one nazwę wskaźników lub w e k t o - r ó w. W wielu wypadkach do wektora, zwłaszcza wektora-cienia, możemy wpisać adres naszego programu, co spowoduje, że będzie on wykonywany zamiast programu OS. By umiejętnie posługiwać możliwościami OS w asemblerze i JM, trzeba poznawać rozmieszczenie i funkcje jego rejestrów.

Omówmy pokrótce podstawowe części składowe OS Atari.

Monitor. Jest to program systemowy wykonywany przy uruchomieniu komputera lub po naciśnięciu klawisza RESET. Monitor inicjuje działanie systemu zarządzania pamięcią i układu wejścia-wyjścia, nadaje początkowe wartości wektorom systemu i sprawdza, czy jest on kompletny.

Monitor wykonuje po włączeniu komputera zimny start skok do podprogramu pod adresem E477 , podczas którego zerowany jest cały RAM z wyjątkiem pierwszych 16 komórek. Naciśnięcie RESET powoduje gorący start (skok pod E474), przy którym pamięć nie jest zerowana, lecz następuje przypisanie ponownie wartości początkowych wektorom systemowym.

Głównym celem systemu zarządzania pamięcią jest właściwe określenie przestrzeni adresowej, w tym górnej i dolnej granicy dostępnego RAM. Istnieje szereg wskaźników dołu i szczytu RAM, które pozwalają użytkownikowi wpływać na mechanizm zarządzania pamięcią.

Struktura wywoływania przerwań. Strukturę przerwań 6502 wykorzystuje się do stosowania wielu rodzajów przerwań. Omówimy je w następnym punkcie.

Wektory systemowe OS. Za ich pośrednictwem użytkownik może wykorzystać do różnych celów podprogramy systemowe OS. Najczęściej czyni się to w procedurach wejścia-wyjścia, posługiwaniu się zegarami i przekazywaniu sterowania innym programom. Dostęp do podprogramów systemowych odbywa się na dwa sposoby: za pośrednictwem skoków do adresów w ROM lub za pośrednictwem rejestrów-cieni w RAM.

Podsystem wejścia-wyjścia. OS zapewnia programiście szerokie możliwości korzystania z urządzeń zewnętrznych. Omówimy je w punkcie 9.8.

Programowanie w czasie rzeczywistym. Wyposażenie OS składa się tu przede wszystkim z zegarów systemowych, sprzętowych, liczących czas w tempie od pół mikrosekundy do paru sekund, oraz programowych odliczających pięćdziesiąte części sekundy.

Zbiory znaków. CHARSET1 i CHARSET2 można wykorzystać bezpośrednio lub redefiniować w RAM.

Pakiet zmienno-pozycyjny. Podprogramy wykorzystują arytmetykę BCD do wykonywania wielu funkcji. Omówimy je w punkcie 9.9.

9.7 Przerwania

Podczas wykonywania programu zachodzą zdarzenia wymagające natychmiastowej reakcji 6502 i komputera. Należy do nich wprowadzanie danych z klawiatury, współpraca ze stacją dyskietek, odświeżanie ładunków w pamięci wykonywane co ułamek sekundy przez specjalny układ i wiele innych zdarzeń. Na wszystkie takie sytuacje w 6502 przewidziano wywołanie p r z e r w a ń. W chwili przerwania wykonanie bieżącego programu ulega czasowemu wstrzymaniu, a wywoływany jest specjalny program obsługi przerwania. Po jego wykonaniu sterowanie przekazywane jest na powrót programowi, który był wykonywany przed przerwaniem.

6502 przewiduje przerwania trzech typów:

- IRQ (interrupt request) - na żądanie użytkownika wektor zawierający adres programu ich obsługi znajduje się pod FFFE hex;

- NMI (non maskable interrupt) - niemaskowalne czyli w zasadzie ustanawiane przez system. Wektor programu obsługi znajduje się pod FFFA hex;

- RESET - przerwanie zimnego lub gorącego startu, wywoływane w szczególności przez naciśnięcie klawisza RESET. Wektor odczytujemy pod FFFC.

Duża część przerwań wywoływana jest przez układy wewnęt-

rzne komputera bez udziału programującego. Jednakże może on również wykorzystać przerwania do własnych potrzeb.

Zasadniczy mechanizm ich obsługi tworzy 6502 z pomocą trzech rozkazów: SEI, CLI i RTI oraz wspomnianych adresów na szczycie ROM.

SEI – set interrupt disable – ustawia znacznik I w rejestrze stanu procesora P. Powoduje to, że od tej chwili wszelkie przerwania IRQ są z a b r o n i o n e. Dlaczego stosuje się ten krok? Po to, by wykonywane w owej chwili przez program czynności związane z ustanowieniem i wykonaniem nowego przerwania nie mogły być zakłócone przez jakiekolwiek inne niespodziewane przerwanie.

Odwrotną czynność, to znaczy włączenie działającego w komputerze systemu przerwań wykonuje rozkaz CLI – clear interrupt disable . Kasuje on znacznik I.

Dla zrozumienia działania RTI konieczna jest dodatkowa informacja. Otóż przy każdym wywołaniu programu obsługi przerwania 6502 automatycznie umieszcza na stosie wartości licznika programu PCH i PCL, tak jak to czyni JSR, ale dodatkowo wstawia na stos także stan rejestru P s p r z e d przerwania. Tak więc w rejestrze P czasowo wstawionym na stos znacznik zakazu przerwań jest skasowany. Dodajmy tu, że w wielu przypadkach celowe jest przechowanie na stosie również wartości A, X i Y sprzed przerwania, co np. przy wywołaniu pustego przerwania pionowego automatycznie wykonuje OS, a w wielu innych wypadkach powinniśmy uczynić my.

Program obsługi przerwania kończyć trzeba rozkazem RTI, który odtwarza dawne rejestry P i PC.

Tak przedstawia się ogólny mechanizm przerwań, jaki tworzy mikroprocesor 6502. Jest to, oczywiście, dopiero ogólną ramą, w którą OS Atari wpisuje szereg bardzo użytecznych dla programującego przerwań.

9.7.1 Rodzaje przerwań w Atari

Jak użytkownik może wykorzystać mechanizm przerwań? Odpowiedź nie jest prosta wobec zróżnicowania przerwań i nieco odmiennych dróg ich obsługi. Najogólniej mówiąc, należy spo-

wodować, by komputer zaczął wykonywać nasz program obsługi przerwania zamiast standardowego. Z reguły wymaga to stosowania języka maszynowego.

W Atari, podobnie jak we wszystkich komputerach opartych na 6502, istnieją trzy typy przerwań: NMI, RESET i IRQ. Stosunkowo najmniej jest do powiedzenia o drugim z nich. Układy komputera dowiadują się o nim, gdy włączamy komputer, naciskamy klawisz RESET lub wywołujemy program zimnego bądź gorącego startu. OS uruchamia wówczas program obsługi tego przerwania, który przewiduje bardzo liczne czynności związane z przygotowaniem komputera do pracy. RESET nie ma wektora, do którego moglibyśmy wpisać nasz program obsługi. Natomiast duże znaczenie ma wektora C-D (12-13 dec). Jeżeli wpiszemy tu adres startu naszego programu, to po RESET nastąpi ponowne jego uruchomienie.

RESET należy do kategorii przerwań NMI i podobnie jak dwa pozostałe obsługiwany jest przez ANTIC.

Przerwania niemaskowalne są zgodnie z nazwą takimi, których nie można zamaskować czyli powstrzymać. Mają one najwyższy priorytet. W przypadku RESET jest to oczywiste, nie można pracować na komputerze, który nie jest do tego gotów. Dwa pozostałe przerwania NMI wiążą się z równie konieczną czynnością wyprowadzania obrazu na ekran. Są to przerwania: listy obrazu display list interrupt - DLI i synchronizacji pionowej czyli pustego przerwania pionowego (vertical blank interrupt - VBI). Są one wywoływane przez system w regularnych odstępach czasu.

Możliwość wykorzystania obu przerwań wynika z tego, że w pierwszym pozostaje kilkadziesiąt cykli, a w drugim nawet do 20 tysięcy cykli na wykonanie przez 6502 programu, który wpiszemy w przerwanie.

Chęć wykorzystania przerwania DLI sygnalizujemy wpisując do listy obrazu rozkaz, którego kodem jest ustawienie najwyższego bitu w jednym z bajtów określających tryb graficzny linii tego trybu. Trzeba poinformować ANTIC, które z przerwań chcemy wykorzystać. Wykonujemy to przez ustawienie odpowiedniego bitu w ważnym rejestrze sterującym NMIEN (NMI enable -

uczynnienie NMI) o adresie D40E (54286). Bity tego rejestru mają następujące znaczenia:

Bit:	7	6	5	4-0
Przerwanie:	DLI	VBI	RESET	nie używane

Dla DLI trzeba ustawić o b a bity b7 i b6 czyli wpisać C0 hex, dla VBI - 40 hex. Rejestr NMIRES w następnej komórce zawierać będzie kopię tych danych. Wpisanie do niego dowolnej wartości kasuje przerwanie.

I wreszcie, niezbędny jest jeszcze jeden ważny krok. W s z y s t k i e przerwania poza RESET mają w RAM swoje wektory czyli dwubajtowe komórki, do których programujący może wpisać adres startu swego programu obsługi przerwania. Musi go kończyć rozkaz RTI zapewniając prawidłowy powrót z przerwania.

Przerwania na żądanie różnią się od NMI tym, że mają niższy priorytet i są sterowane inaczej: przez POKEY i ewentualnie PIA.

Żądanie przerwania IRQ zgłaszamy w naszym programie rozkazem 6502 SEI. Po tym sygnale OS niezwłocznie przystępuje do ustalenia, o jakie przerwanie chodzi. Należy wziąć pod uwagę, że OS ma własny program obsługi IRQ, który wykonuje odpowiednie działania w przypadku, gdy przerwania żąda np. klawiatura lub OS przy przesyłaniu danych szyną szeregowego wejścia-wyjścia. Adresy odpowiednich programów obługi znajdują się w dwubajtowych wektorach, podobnie jak w przypadku NMI. Programy obsługi przerwań zegarów POKEY-a są bardzo proste: składają się tylko z rozkazów: PLA, RTI zapewniających powrót z przerwania.

Podstawowa informacja o rodzaju przerwania zawarta jest w rejestrze IRQEN o adresie D20E. Ustawienie przez nas bitu w tym rejestrze powoduje, że OS przystępuje do obsługi danego przerwania. Zestawmy w jednej tabeli informacje o wszystkich przerwaniach NMI i IRQ. Adresy dotyczą pierwszego bajtu wektora. W rubryce "Bit" podano odpowiedni bit z rejestru NMIEN lub IRQEN. W ostatniej rubryce wskazano, kto wykorzystuje przerwanie, przy czym U oznacza "użytkownik".

Przerwanie	Wektor	hex	Typ	Bit	Kto?
Listy obrazu - DLI	VDSLST	200	NMI	6,7	U
RESET	nie ma		NMI	5	OS
Puste pionowe - VBI zwykłe	VVBLKI	222	NMI	6	OS,U
Puste pionowe - VBI opóźnione	VVBLKD	224	NMI	6	OS,U
Dane na wejściu szeregowym gotowe	VSERIN	20A	IRQ	5	OS
Potrzebne dane na wyjściu szeregowym	VSEROR	20C	IRQ	4	OS
Przesył na wyjściu szeregowym wykonany	VSEROC	20E	IRQ	3	OS
Zegar POKEY-a 1 odlicza w dół do 0	VTIMR1	210	IRQ	0	U
Zegar POKEY-a 2 j.w.	VTIMR2	212	IRQ	1	U
Zegar POKEY-a 4 j.w.	VTIMR4	214	IRQ	2	U
Naciśnięty klawisz BREAK	BRKKY	236	IRQ	7	OS
Naciśnięty inny klawisz	VKEYBD	208	IRQ	6	OS

Rys. 9.1 Przerwania w Atari

Rozszerzmy niektóre informacje zawarte w tabeli. Kilka przerwań ustanawianych jest przez OS. Trzy z nich dotyczą obsługi szeregowego wejścia-wyjścia i mają na celu synchronizację przesyłu danych konieczną wobec faktu, że urządzenia zewnętrzne pracują wolniej niż układy komputera. W tym wypadku ingerencja użytkownika byłaby szkodliwa.

Posługiwanie się w przerwaniach zegarami POKEY-a możliwe jest dzięki temu, że gdy licząc malejąco dochodzą do zera, uruchamiamy jest program, którego adres startu zapisze się w odpowiednim wektorze na stronia 2.

Przerwania klawiatury i programowe BREAK ustanawia OS. Może je wykorzystać programujący, co przedstawia przykład podany w podpunkcie 9.7.3.

Istnieją ponadto dwa przerwania dostępne na szynie szeregowej, lecz nie wykorzystywane przez OS, planowane dla późniejszych rozszerzeń.

9.7.2 Przerwania w tworzeniu obrazu

Dwa przerwania niemaskowalne - DLI i VBI - dają programującemu możliwość wtrącenia w tok wykonania programów ANTIC-u dodatkowych obliczeń wykonywanych przez 6502.

Pierwsze z przerwań, a mianowicie przerwanie listy obrazu DLI, jest trudne do realizacji ze względu na ostre restrykcje czasowe. Można w nim wykonać stosunkowo nieduże zadania, np. kolorystyczne wzbogacenie obrazu, nieznaczne usprawnienie grafiki graczy-pocisków, manewrowanie paru zbiorami znaków itp.

Oto dwa przykładowe programy wykonujące w DLI nieduże zadania: pierwszy powoduje druk tekstu "do góry nogami" w dolnej połowie ekranu, drugi - zmianę barwy tej połowy. Oba podprogramy przeznaczone są do wywołania z Basicu:

X=USR(1552)

Mają one wspólną część sterującą pod adresem 610 hex.

Pierwszy program (dane w hex):

```
 *= 600           Początek programu obsługi przerwania
SDLSTL = 230      Wektor początku listy obrazu
VDSLST = 200      Wektor przerwań DLI
CHACTL = D401     Rejestr kontroli wyświetlania znaków
NMIEN  = D40E     Podstawowy rejestr NMI
ZP     = CB       Wskaźnik na stronie zerowej
    PHA           Zawartość A na stos
    LDA #4        Wartość powodująca odwrócenie znaków
    STA CHACTL    Rejestr, który to wykonuje
    PLA           Odtworzenie A
    RTI           Powrót z przerwania
 *= 610           Adres części sterującej
    PLA           Liczba parametrów USR
    LDA SDLSTL    Przeniesienie SDLSTL na stronę zerową
    STA ZP
    LDA SDLSTL+1
    STA ZP+1
    LDY #10       Przesunięcie od początku DL
    LDA #82       Rozkaz przerwania ANTIC-u, tryb 2
```

```
STA (ZP),Y          Wpisanie rozkazu do listy obrazu
LDA #0              Wpisanie adres 600 do VDSLST
STA VDSLST
LDA #6
STA VDSLST+1
LDA #C0             Kod przerwania DLI ...
STA NMIEN           wpisany do NMIEN
RTS                 Powrót do Basicu.
```

A teraz drugi, minimalnie dłuższy program obsługi przerwania, do którego trzeba dopisać tę samą część sterującą od 610 hex.

```
    *= 600
VSYNC  = D40A       Rejestr ten - wait for horizontal syn-
                    chronization - umożliwia wykonanie prog-
                    ramu dopiero przed rozpoczęciem następnej
                    linii ekranowej
COLPF2 = D018       Jeden z rejestrów kolorów GTIA
    PHA             Jak poprzednio
    LDA  DE         Kolor żółto-zielony, duża jasność
    STA WSYNC
    STA COLPF2
    PLA
    RTI
```

Wprowadzenie wartości koloru najpierw do WSYNC "zamraża" na chwilę 6502 zapobiegając zakłóceniom w określeniu barwy.

Przykłady te ilustrują stosunkowo skromne możliwości DLI. Na tym tle lepiej widać, jak bardzo użyteczne jest drugie z wspomnianych przerwań: puste przerwanie pionowe. Jest ono bez wątpienia najważniejszym dla programisty przerwaniem w Atari.

Co 1/50 sekundy w systemach PAL i SECAM, a co 1/60 sek w stosowanym w USA systemie NTCS wiązka elektronów wywołująca fluoryzację ekranu telewizora lub monitora po dojściu do prawego dolnego rogu ekranu przeskakuje w jego lewy górny róg. W tym ułamku sekundy, choć właściwości oka nie pozwalają tego dostrzec, ekran jest pusty. I na ten ułamek sekundy komputer przerywa wysyłanie danych obrazu, a zatem ANTIC nie wykonuje żadnych zadań. Możliwe jest wykorzystanie tego czasu na pracę

6502. Wprawdzie wedle potocznych miar puste przerwanie pionowe trwa nader krótko, mimo to można w nim zmieścić parę tysięcy cykli zegarowych pracy 6502.

Możliwe jest ponadto wydłużenie tego czasu przez zastosowanie tzw. przerwania opóźnionego (ang. deferred). Nieco rozbieżne są dane co do wielkości dostępnej dla 6502 pauzy czasowej. W "De Re Atari" [6] mówi się o ok. 2 tysiącach cykli w przerwaniu zwykłym i ok. 20000 w opóźnionym. Gary R. Lecompte w "Third Book of Atari" [9] pisze, że w jednym przerwaniu opóźnionym zależnie od trybu graficznego i innych przerwań pozostaje ok. 7980 cykli. Tak czy owak w puste przerwanie pionowe można wpisać stosunkowo duży fragment programu. Ciekawym przykładem jest pod tym względem przedstawiona przez Lecompte'a gra "Laser Gunner II", w której w obrębie przerwań synchronizacji pionowej wykonuje się program wywołujący ruch obiektów na ekranie. Program ten obejmuje ponadto 60 rozkazów i jest wykonywany 50 razy na sekundę. Łatwo obliczyć skalę zadań możliwych do wykonania w ten sposób. Sprawia to, że VBI są w rękach wytrawnego programisty potężnym środkiem usprawnienia programów.

W obrębie VBI można zaprogramować bardzo różnorodne zadania. Jednym z nich jest wykonywanie znacznych zmian na ekranie w sposób płynny i szybko. Inne zastosowanie wiąże się z wytwarzaniem równolegle do obrazu rozmaitych efektów dźwiękowych. Bardziej skomplikowanym rozwiązaniem jest równoległe wykonywanie dwóch programów - jednego w przerwaniach, a drugiego podczas normalnej pracy procesora. Wymaga to starannego rozdzielenia danych dla obu programów.

Program obsługi VBI przez OS ma pewne specyficzne cechy. Gdy OS rozpczna to przerwanie, zawartość rejestrów A, X i Y wstawiana jest na stos. Następnie odczytywany jest wektor VVBLKI (222 hex) i wywoływany jest program spod znajdującego się tam adresu. Normalnie jest to program obsługi przerwania znajdujący się pod etykietą SYSVBV (adres E45F), który wykonuje wszystkie czynności przewidziane dla VBI, jak zwiększenie zegara czasu rzeczywistego, skopiowanie rejestrów-cieni i in. Następnie program wykonuje skok pod adres zapisany w

wektorze opóźnionego przerwania VVBLKD (224 hex), co normalnie oznącza skok do XITVBV pod E462 powodujący zakończenie przerwania.

Programista może "wtargnąć" do tego mechanizmu w dwóch punktach: w wektorach VVBLKI lub VVBLKD. Wymaga to decyzji co do wyboru punktu. W wielu wypadkach nie jest to zbyt istotne, w paru jednak ma znaczenia. Są to sytuacje, w których chcemy zmienić wartości rejestrów. Korzysta z nich również program obsługi przerwania, toteż musimy to uczynić po nim. Drugi przypadek szczególny powstaje, gdy nasz program obsługi VBI jest długi. Wprowadzając go na początku opisanego ciągu działań ryzykowalibyśmy, że jego część będzie wykonywana po zakończeniu VBI z nieprzewidzianymi skutkami. W obu wypadkach należy wybrać przerwanie opóźnione (deferred).

Gdy wybór został dokonany, musimy umieścić nasz program obsługi w pamięci i do odpowiedniego wektora - VVBLKI lub VVBLKD - wpisać jego adres, a następnie zmienić jeszcze odpowiedni wektor programu OS. Wszystko to, przypomnijmy, realizowane będzie w olbrzymim tempie, toteż zanim wpiszemy drugą połowę adresu, system może runąć. Rozwiązanie polega na wykorzystaniu specjalnego programu OS pod nazwą SETVBV (set vertical blank vector - ustanów wektor pustego przerwania pionowego) pod adresem E45C. Aby to wykonać należy:

- załadować do rejestru Y młodszy bajt adresu, ...
- a do rejestru X starszy bajt adresu naszego programu obsługi VBI;
- załadować do akumulatora 6 dla natychmiastowego lub 7 dla opóźnionego VBI;
- wykonać skok JSR SETVBV.

Nowy program obsługi VBI zacznie być pomyślnie wykonywany w ciągu najwyżej 1/50 sek. Jednakże aby jego zakończenie było pomyślne, musi on we właściwe miejsce powrócić do opisanego wcześniej programu standardowego. Dlatego program nasz musi kończyć się skokiem do wektora n a s t ę p n e g o po miejscu naszego odgałęzienia: do SYSVBV przy natychmiastowym, a do XITVBV przy opóźnionym VBI.

Ilustruje to poniższy krótki podprogram powodujący błys-

kanie kursora, a także znaków na inwersie. Jego wywołanie:

X=USR (1664)

Podprogram napisany jest w hex.

```
SETVBV = E45C
VVBLKD = 224
XITVBV = E462
CHACT  = 2F3
RTCLOK = 14
        *= 680
        PLA             Zdjęcie liczby parametrów USR
        LDY #90         Młodszy bajt adresu naszego programu obsłu-
                        gi przerwania w rejestrze Y
        LDX #6          Starszy bajt w X
        LDA #7          W A 7 dla przerwania opóźnionego
        JSR SETVBV
        RTS             Powrót do Basicu
        *= 690          Start programu obłsugi VBI
        LDA RTCLOK      Wartość zegara
        AND #10         Wyodrębnia bit b4
        LSR A           Przesunięcie na b3
        LSR A           ... na b2
        LSR A           ... na b1
        STA CHACT       Wpisanie do CHACT
        JSR XITVBV
```

Gdy w CHACT jest 0, kursor jest niewidzialny, znaki mimo naciśnięcia klawisza LOGO wyświetlane są normalnie. Gdy w CHACT jest 2 - kursor i znaki na inwersie są widzialne. Dane z zegara powodują wpisanie do CHACT na zmianę 2 lub 0. Powoduje to błyskanie kursora i znaków na inwersie.

Program będzie działać identycznie, gdy zastosujemy parę VVBLKI (6 w akumulatorze) i SYSVBV zamiast VVBLKD i XITVBV. Wykonajmy taką próbę dla nabycia wprawy w posługiwaniu się nader użytecznym pustym przerwaniem pionowym.

9.7.3 Przykład: przerwania klawiatury

Przerwania klawiatury i programowe BREAK ustanawia OS. Może je wykorzystać programujący. Klawisz BREAK może wywoływać wykonanie programu, którego adres zapiszemy w wektorze BRKKY. Przerwania związane z naciśnięciem innych klawiszy mają także wektor. Jego wykorzystanie ilustruje poniższy program stanowiący zmodyfikowaną wersję propozycji z "De Re Atari".

Uważna analiza programu pozwala zorientować się, w jaki sposób i w jakiej kolejności stosować należy rozkazy 6502 SEI CLI i RTI oraz jak zmieniać adresy w wektorach, by osiągnąć wykorzystanie przerwania do naszych celów. W danym wypadku rejestry IRQEN i VKEYBD, a także VIMIRQ wykorzystane zostały do wprowadzenia własnego programu powodującego, że przestają działać klawisze CONTROL i BREAK. Rejestr VIMIRQ zawiera po inicjalizacji adres programu obsługi IRQ. Przykładowy program pokazuje, jak można go "ukraść" do naszych potrzeb. W zapisie zastosowano konwencję asemblera z pominięciem numerów linii.

```
POKMSK = $0010
KBCODE = $0209
VKEYBD = $0208
IRQEN  = $020E
IRQST  = IRQEN
VMIRQ  = $0216
       *= $600        Początek programu na stronie 6
START SEI             Wyłącza przerwania IRQ
       LDA VMIRQ      Zastępuje wektor przerwań IRQ ...
       STA NBRK+1     własnym wektorem przez modyfikację adresu
       LDA VMIRQ+1
       STA NBRK+2     Wszystkie IRQ kierowane będą pod NBRK
       LDA #<IRQ
       STA VMIRQ      W wektorze VMIRQ umieszczamy adres ...
       LDA #>IRQ
       STA VMIRQ+1    naszego programu obsługi IRQ
       LDA VKEYBD
       STA JUMP+1     Modyfikacja adresu JMP pod adresem JUMP
       LDA VKEYBD+1
```

```
        STA JUMP+2
        LDA # REP
        STA VKEYBD        Z kolei wektor VKEYBD otrzymuje adres ...
        LDA #>REP         naszych przerwań klawiatury
        STA VKEYBD+1
        CLI               Ponowne uruchomienie systemu przerwań IRQ
        RTS
        *= $639           Tu początek tworzonego przez nas programu
REP     LDA KBCODE        Wszystkie IRQ klawiszy przyjdą tu
        AND #$80          Czy klawisz CONTROL naciśnięty?
        BEQ JUMP          Jeżeli nie (AND nie dało zera), idź dalej
        PLA               Jeżeli tak, ignoruj klawisz CONTROL
        RTI
JUMP    JMP JUMP          Wywołuje dawne IRQ klawiszy. Modyfikacja!
IRQ     PHA               Wszystkie IRQ przyjdą tu
        LDA IRQST         Sprawdza status IRQ - czy BREAK naciśnięty?
        BPL BREAK         Jeżeli tak, odgałęzienie
        PLA               Jeżeli nie
NBRK    JMP NBRK          wywołuje dawny wektor IRQ. Modyfikacja!
BREAK   LDA #$7F          Tu unieruchomienie BREAK
        STA IRQST
        LDA POKMSK
        STA IRQEN
        PLA
        RTI               Powrót, jakby nie było naciśnięcia BREAK
        *= $02E2
        .WORD START
```

Program wykonuje manewry na wektorach przerwań, o których była mowa. Wyjaśnienia wymaga rola IRQST i IRQEN. Wpisanie do tej komórki, a także do POKMSK wartości $7F wyłącza klawisz BREAK. Równocześnie komórka ta podaje status przerwania IRQST .

Wpisanie adresu startu $600 pod $2E2 powoduje natychmiastowe uruchomienie programu po wgraniu go z dyskietki. Można rzecz prosta, dopisać na początku PLA i uruchomić program z Basicu.

9.8 Wejście-wyjście

Komputer nie może skutecznie wykonywać zadań bez kontaktu ze światem zewnętrznym. Decyduje to o znaczeniu problematyki wejścia-wyjścia. O niektórych kwestiach z tego zakresu mówiliśmy, w tym zwłaszcza o wyprowadzaniu obrazu na ekran. Była to część szerszego zagadnienia właściwej organizacji wejścia-wyjścia. Nowoczesnym systemom operacyjnym stawia się w tej dziedzinie następujące wymogi:

- przesył danych powinien być uniezależniony od urządzenia;
- struktura wejścia-wyjścia powinna umożliwiać przesył danych zarówno pojedyńczymi bajtami, jak i ich ciągami oraz rekordami;
- należy zapewnić równoległy dostęp do kilku urządzeń lub plików jednocześnie;
- jednolity powinien być system kontroli błędów;
- powinno być możliwe dołączenie nowych manipulatorów czyli programów obsługi urządzeń.

Podsystem wejścia-wyjścia w OS Atari zaprojektowano z uwzględnieniem wszystkich tych wymogów. Zapewnia on dużą łatwość programowania obsługi wejścia-wyjścia w asemblerze. Ze względu na konieczność zapewnienia dużej szybkości przesyłu danych jest to język podstawowy w tej dziedzinie.

Przyjęte rozwiązania oparto na jednolitych, niezależnych od urządzeń blokach sterowania wejściem-wyjściem (ang. input -output control blocks - IOCB). Jest osiem IOCB. Każdy z nich może zawierać zestaw danych niezbędnych do wykonania pełnej pojedyńczej operacji We-Wy. Z pomocą IOCB można nie troszcząc się o różnice techniczne i nie mozoląc nad zapamiętywaniem szczegółów powierzyć systemowi operacyjnemu wykonanie operacji przesyłu danych do i z tak różnych urządzeń, jak monitor, magnetofon, stacja dyskietek czy drukarka.

Obok wspomianych ośmiu IOCB w skład systemu wchodzą: pomocniczy blok na stronie zerowej zwany ZIOCB, blok sterowania urządzeniem (device control block - DCB) oraz 4-bajtowa tablica dla przesyłu szeregowego, którą użytkownik nie powinien się posługiwać (command frame buffer - CFB).

W stosunku do swego znaczenia są to jednostki objętościowo niewielkie. IOCB i ZIOCB liczą po 16 bajtów każdy, a DCB 19 bajtów. Mają one następujące adresy w hex:

IOCB0	340-34F	IOCB6	3A0-3AF
IOCB1	350-35F	IOCB7	3B0-3BF
IOCB2	360-36F		
IOCB3	370-37F	ZIOCB	20-2F
IOCB4	380-38F	DCB	300-30B
IOCB5	390-39F	CFB	23A-23D

IOCB0 jest wykorzystywany przez edytor ekranu. IOCB6 poza trybem graficznym 0 Basicu służy do wyprowadzania na ekran i trzeba go wówczas zamknąć przed użyciem. IOCB7 wykorzystywany jest w obsłudze drukarki.

Na wyższym piętrze systemu znajdują się dwa programy OS: centralny program systemowy We-Wy CIO oraz odmiennie wykonujący zadania program We-Wy szeregowego SIO. CIO jest niedużym programem liczącym ok. 800 bajtów, który jedynie kieruje dane z IOCB do manipulatora czyli programu obsługi wskazanego urządzenia.

I wreszcie w funkcjonowaniu tego mechanizmu dużą rolę odgrywa tablica adresów manipulatorów urządzeń (ang. handler address table - HATABS), według której CIO odnajduje właściwy manipulator. HATABS zajmuje 34 bajty w obszarze 31A-33D. Składa się ona z trzybajtowych części, z których każda zawiera kod ASCII jednoliterowej nazwy urządzenia oraz adres programu jego obsługi czyli manipulatora. OS wpisuje do tablicy następujące manipulatory: P - drukarka, C - magnetofon, E - edytor ekranu, S - ekran, K - klawiatura. W chwili uruchomienia komputera w HATABS nie ma wpisu stacji dyskietek D. W OS znajduje się jedynie bardzo zwięzły program obsługi takiej stacji. Dopiero po wgraniu dyskowego systemu operacyjnego w HATABS pojawia się litera i adres manipulatora. Podobnie jest z manipulatorem modułu RS232.

Jednakże w HATABS nawet po wprowadzeniu D i R są jeszcze cztery wolne miejsca dla nazw i adresów manipulatorów urządzeń. Spełniony jest zatem postulat rozszerzalności systemu.

Programiści wykorzystali go opracowując m.in. manipulatory dla urządzeń M i N. Pierwsze stanowi utworzony w RAM obszar swoistej dyskietki wewnętrznej, drugie jest tzw. null handlerem - urządzeniem, które ... nic nie robi. Może ono być pomocne przy uruchamianiu programów.

W programowaniu wejścia-wyjścia podstawowe znaczenie ma dobra znajomość struktury IOCB i sposobów korzystania z nich, jest to bowiem najważniejsza droga zorganizowania przesyłu danych. Z Basicu znamy ją w postaci tzw. kanałów, które można otworzyć i odpowiednio zaprogramować. Tę samą rolę w asemblerze i JM pełnią IOCB. Wszystkie IOCB mają jednakową strukturę. Przedstawmy ją numerując bajty od 0 do 15. W asemblerze stosuje się dla nich etykiety. Oczywiście, bajt 0 w IOCB0 mieć będzie adres 340, w IOCB1 - 350 itd.

Etykieta	Nr bajtu	Opis
ICHID	0	Wpisuje OS. Indeks w HATABS nazwy urządzenia dla pliku bieżąco otwartego
ICDNO	1	Wpisuje OS. Nr urządzenia, np. 1 dla D1;
ICCOM	2	Komenda określająca rodzaj czynności
ICSTA	3	Wpisuje OS. Status urządzenia. 1 gdy operacja przebiegła pomyślnie, numer błędu w przeciwnym wypadku
ICBAL/H	4 i 5	Dwubajtowy adres buforu dla przesyłu danych lub adres nazwy pliku
ICPTL/H	6 i 7	Wpisuje OS. Adres minus 1 programu przesłania bajtu do danego urządzenia
ICBLL/H	8 i 9	Długość buforu wskazanego w ICBA (4-5). OS zmniejsza ją o 1 po każdym przesłanym bajcie
ICAX1	A	Bajt pomocniczy 1. Używany w OPEN do określania rodzaju dostępu
ICAX2-ICAX6	B-F	Bajty pomocnicze. ICAX3-ICAX5 wykorzystywane są w operacjach NOTE i POINT. Poza tym mają niejednolite przeznaczenie.

Jak może wykorzystać programujący podane tu informacje? Przede wszystkim powinien pamiętać, że nigdy nie należy inge-

rować w bajty 0, 1, 3, 6 i 7 zarezerwowane dla OS, który wykorzystuje je przy przesyle danych. Spośród pozostałych bajtów największe znaczenie ma ICCOM. Komenda tu wpisana określa sposób wykorzystania innych komórek IOCB.

Korzystanie z IOCB należy rozpocząć od jego otwarcia komendą OPEN o kodzie w ICCOM 03. Równocześnie w ICAX1 podaje się rodzaj zadania, któremu ma służyć IOCB. Oto znaczenia poszczególnych wpisów:

ICCOM	ICAX1	
3	4	Read. Urządzenie lub plik otwarte dla czytania
3	8	Write. Otwarte dla zapisu
3	12	Update. Otwarte dla czytania i zapisu. W edytorze ekranu oznacza to wprowadzenie z klawiatury i wyprowadzenie na ekran
3	6	Czytanie tylko katalogu dyskietki
3	9	Write-append. Dołączenie danych do zbioru na dyskietce

Przy komendzie OPEN w ICBAL/H musi znajdować się adres, pod którym znajduje się nazwa pliku.

Po dokonaniu wszystkich niezbędnych wpisów w celu wykonania komendy wywołujemy CIO. Wykonuje się w tym celu skok JSR CIOV do wektora CIO pod adresem E456, przy czym w rejestrze X musi znajdować się numer IOCB pomnożony przez 16 czyli 10 hex. Sekwencja rozkazów jest zwykle następująca:

```
LDX #IOCBNUM    Numer pomnożony przez 16!
JSR CIOV        Skok do podprogramu pod E456
BMI ERROR       Jeżeli wynik ujemny, powstał błąd.
```

Skok do sekwencji obsługi błędów z pomocą BMI wynika z tego, że w przypadku błędu CIO umieszcza w rejestrze Y jego numer, większy lub równy 80 hex czyli 128 dec. Jeżeli operacja przebiegła pomyślnie, CIO przesyła 1. Sprawdzenie tego jest praktycznie konieczne dla zapobieżenia zawieszeniu się programu.

Inne komendy w ICCOM odpowiadają przede wszystkim funkcjom, jakie wykonuje CIO. Przy każdej z nich ICBA powinien

znajdować się adres buforu dla wprowadzania lub wyprowadzania danych. Buforem może być w zasadzie dowolne miejsce w RAM. W ICBL powinna znajdować się długość buforu. Oto funkcje i ich kody w ICCOM:

ICCOM	
3	OPEN.
C	CLOSE. Zamknięcie pliku lub urządzenia
5	GET RECORD. Komenda ta zapewnia odczytanie z otwartego urządzenia rekordu czyli ciągu bajtów o wskazanej długości. Taką formę ma np. czytanie sektorów dyskietki. Przy czytaniu w edytorze kończy się ono po napotkaniu znaku RETURN; kod 9B-155
7	GET BYTES. Czytanie N bajtów, tzw. rekordu binarnego
9	PUT RECORD. Zapis rekordu. Czynność odwrotna do GET RECORD
B	PUT BYTES. Zapis N bajtów

Inne funkcje:

D	STATUS. Komenda ta podaje stan wskazanego urządzenia, którego adres zawiera ICBA
różne	SPECIAL. Są to wywołania manipulatorów specyficznych dla niektórych urządzeń. W szczególności w przypadku stacji dyskietek możliwe są poniższe komendy, których kody odpowiadają numerom funkcji XIO w Basicu.

ICCOM hex	dec	
20	32	Rename. Zmiana nazwy pliku
21	33	Erase. Skasowanie pliku
23	35	Protect (lock). Ochrona przed zapisem
24	36	Unprotect (unlock). Zniesienie ochrony
25	37	Point. Odpowiada analogicznej funkcji w Basicu
26	38	Note. Jak wyżej
FE	254	Format. Formatowanie całej dyskietki.

By wykonać którąkolwiek z tych komend, należy wywołać CIO we wskazany wcześniej sposób. W następnym punkcie rozdzia-

łu znajduje się przykład programu dość przejrzyście wyjaśniającego ten mechanizm.

Elementem podsystemu wejścia-wyjścia jest blok kontroli urządzenia DCB i SIO. Kolejne bajty w DCB mają następujące funkcje:

Etykieta	Bajt	Opis
DDEVIC	0	Wpisuje OS. Identyfikator urządzenia
DUNIT	1	Numer urządzenia bieżąco używanego. Określa użytkownik
DCOMND	2	Komenda ustalana przez użytkownika lub manipulator przed wywołaniem SIO
DSTATS	3	Stan operacji
DBUFL/H	4-5	Adres buforu źródłowego lub docelowego. Wpisuje użytkownik
DTIMLO	6	Timeout w sekundach czyli czas oczekiwania na pomyślne połączenie się z urządzeniem
DUNUSE	7	Bajt nie używany
DBYTL/H	8-9	Wpisuje manipulator. Liczba przesłanych bajtów tów
DAUX1/2	A-B	Bajty pomocnicze. W większości operacji numery sektorów dyskietki

Kody komend w DCOMND są m.in. następujące w hex:

"!"	21	formatowanie całej dyskietki
"P"	50	wpisanie sektora bez weryfikacji
"R"	52	czytanie sektora
"S"	53	żądanie statusu
"W"	57	wpisanie sektora z weryfikacją.

Do wywołania SIO po wpisaniu danych do DCB służy wektor SIOV - E459. Może go wykorzystać programujący.

Należy jednak przypomnieć, że niezależny od urządzeń podsystem CIO pozwala komunikować się ze wszystkimi podstawowymi urządzeniami i w razie potrzeby wywołuje SIO.

Przy wykonywaniu operacji wejścia-wyjścia częste są przypadki, gdy urządzenia zewnętrzne i komputer mają rozmaitą szybkość pracy. W takim wypadku istotną rolę odgrywa mechanizm buforów. Jest on szeroko stosowany w systemie operacyjnym Ata-

ri. Przesyłane lub odbierane dane są przejściowo gromadzone w buforze. Czasem bufor jest po prostu miejscem ostatecznego przeznaczenia danych. Elementem buforowania jest l i c z n i k przesyłanych bajtów oraz bajt lub bit s t a n u wejścia-wyjścia, dostarczający informacji czy kolejny bajt w buforze jest gotów do przesłania lub czy został odebrany.

Mechanizmy wejścia-wyjścia dostępne w asemblerze i JM są bogatsze niż w Basicu, który np. nie rozporządza możliwością przesyłu bloków bajtów. Przemawia to za przyswajaniem i doskonaleniem umiejętności programowania We-Wy w asemblerze. Można w ten sposób również zwiększać efektywność programów w Basicu.

9.9 Arytmetyka zmiennopozycyjna

Stosowanie arytmetyki zmiennopozycyjnej ułatwia operowanie liczbami rzeczywistymi, a ściślej ich przybliżeniami w postaci skończonych ułamków dziesiętnych. W systemiem operacyjnym Atari znajduje się pakiet programów, umożliwiających posługiwanie się taką arytmetyką. Z pakietu tego korzysta Atari Basic. Możliwość posługiwania się ułamkami w sposób względnie prosty stanowi udogodnienie. Płaci się za nie cenę w postaci wydatnego zmniejszenia szybkości obliczeń w stosunku do tej, jaką osiągamy w arytmetyce całkowitoliczbowej.

W Atari zastosowano odmienną niż w wielu innych komputerach reprezentację liczb zmiennopozycyjnych. Każda zajmuje 6 bajtów. W pierwszym znajduje się wykładnik, którego najstarszy bit jest bitem znaku (0 dla liczb dodatnich). Pozostałe bity pierwszego bajtu stanowią wykładnik potęgi liczby 100 (a nie 10) zwiększony o 64. Zwiększenie to pozwala na uzyskanie szerokiej skali dodatnich i ujemnych wykładników bez korzystania z bitu znaku. Zakres dopuszczalnych liczb sięga od 10^{-98} do 10^{+98}.

Mantysa zajmuje pozostałych pięć bajtów liczby i zapisana jest w nich w kodzie BCD, co się nieczęsto zdarza. Wybór takiej reprezentacji pozwolił zmniejszyć błędy zaokrągleń. Mantysa jest zawsze znormalizowana w taki sposób, że jej pierwszy bajt nie równa się zeru. Za bajtem tym implikuje się

istnienie kropki dziesiętnej. Toteż gdy wykładnik jest mniejszy niż 64 40 hex , wskazuje to na liczbę o wartości bezwzględnej mniejszej niż 1.

Oto przykłady liczb i ich reprezentacji w hex:

0.02 = $2x100^{-1}$	3F 02 00 00 00 00	/wykł.=40-1/
-0.02 = $-2x100^{-1}$	BF 02 00 00 00 00	/wykł.=80+40-1/
28.0 = $28x100^{0}$	40 28 00 00 00 00	/wykł.=40+0/
-462871 = - $46.0312x100^{2}$	C2 46 03 12	
-392871 = $-39.2871x100^{2}$	C2 39 28 71 00 00	/wykł.=80+40+2/

Zero jest traktowane jako przypadek szczególny: wykładnik i mantysa są równe 0. Test wyniku 0 można zatem przeprowadzać na wykładniku, jak i mantysie.

W tej reprezentacji granice potęgi wyrażonej w wykładniku rozciągają się od -98 do +98.

Pakiet zmiennopozycyjny (ang. floating point - FP) zajmuje obszar D800-DFFF hex. OS automatycznie wyłącza go w czasie operacji We-Wy z urządzeniami zewnętrznymi, a po ich zakończeniu włącza go. Oznacza to, że w operacjach tych program, a więc również interpretator Basicu, nie może korzystać z liczb zmiennopozycyjnych. Pakiet wykorzystuje do swej pracy dwa obszary RAM: D4-FF na stronie zerowej i 57E-5FF na stronie 5. Dwa 6-bajtowe pseudorejestry FRO (adresy D4-D9) i R1 (E0-E5) służą do przechowywania liczb zmiennopozycyjnych. Dwubajtowy wskaźnik FPLPTR (FC-FD) wskazuje adres przetwarzanej liczby zmiennopozycyjnej.

Istnieją dwa bufory do lokowania ciągów cyfr w kodzie ATASCII: dla ciągów wprowadzanych adres zawiera wskaźnik INBUFF (F3-F4), a ciągi wyprowadzane lokowane są w 128-bajtowym buforze LBUFF pod adresami 580-5FF. Istnieje ponadto wskaźnik CIX (F2) zapisujący pozycję przetwarzanej cyfry.

Jeżeli w asemblerze chcemy posłużyć się pakietem FP, tok postępowania jest na ogół następujący. Łańcuch cyfr reprezentujący jedną z liczb musi być zapisany w buforze gdziekolwiek w pamięci. Wskaźnik INBUFF musi wskazać początek tego ciągu. CIX trzeba wyzerować. Wywołuje się program AFP przekształcający ciąg cyfr w liczbę w reprezentacji FP. Efektem jest umieszczenie liczby w FRO, skąd jest pobierana do operacji. Po jej

zakończeniu wynik umieszczany jest we FR0. Teraz program FASC przekształca wynik w ciąg cyfr i umieszcza w LBUFF.

Aby przekształcić liczbę 16-bitową w FP, trzeba umieścić ją w najniższych bajtach FR0 (D4 i D5) i wykonać skok do programu IFP, który przekształci ją w zmiennopozycyjną, a tę pozostawi również we FR0. Odwrotną czynność wykonuje program FPI. Tablica na rysunku 9.1 zawiera zestawienie wszystkich programów pakietu FP z ich nazwami, adresami w ROM, miejscem pozostawiania wyniku i orientacyjnym maksymalnym czasem wykonania w mikrosekundach. Tablica jak i przykładowy program oparte zostały na [5].

Nazwa	Adres	Funkcja		
AFP	D800	ATASCII na FP	FR0	3500
FASC	D8E6	FP na ATASCII	LBUFF	950
IFP	D9AA	Liczba całkowita na FP	FR0	1330
FPI	D9D2	FP na liczbę całkowitą	FR0	2400
FSUB	DA60	Odejmowanie FR0-FR1	FR0	740
FADD	DA66	Dodawanie FR0+FR1	FR0	710
FMUL	DADB	Mnożenie FR0 razy FR1	FR0	12000
FDIV	DB28	Dzielenie FR0 przez FR1	FR0	10000
FLD0R	DD89	Ładowanie liczby FP z użyciem X i Y	FR0	70
FLD0P	DD8D	Ładowanie liczby FP z użyciem FLPTR	FR0	60
FLD1R	DD98	Ładowanie liczby FP z użyciem X i Y	FR1	70
FLD1P	DD9C	Ładowanie liczby FP z użyciem FLPTR	FR1	60
FSTOR	DDA7	Zapisanie liczby FP z użyciem X i Y	FR0	70
FSTOP	DDA8	Zapisanie liczby FP z użyciem FLPTR	FR0	70
FMOVE	DDB6	Przenosi FR0	FR1	60
PLYEV	DD40	Ewaluacja wielomianowa	FR0	88300
EXP	DDC0	Liczba e do potęgi FR0	FR0	115900
EXP10	DDCC	Liczba 10 do potęgi FR0	FR0	108800
LOG	DECD	Logarytm naturalny	FR0	136000
LOG10	DED1	Logarytm dziesiętny	FR0	125400
ZFR0	DA44	Wyzerowanie FR0	FR0	80
AF1	DA46	Wyzerowanie rejestru w X	różne	80

Rys. 9.2 Programy pakietu FP

Przykładowy program przedstawiony niżej ciekawy jest z paru powodów. Czyta on dwie liczby, które podamy w postaci ciągów znakowych z edytora, dokonuje ich konwersji na liczby

zmiennopozycyjne, odejmuje pierwszą od drugiej, a wynik umieszcza w zdefiniowanym przez użytkownika buforze FTEMP oraz wyświetla go na ekranie.

Mimo pozornie niewielkiego znaczenia samej operacji poznajemy tu szereg istotnych czynności, w tym wykonanie przez CIO przeniesienia ciągu znaków z edytora ekranu E do buforu, a także wyświetlenie wyniku na ekranie. Pierwsza część programu przedstawia sposób wykorzystania w asemblerze kilku programów z pakietu FP. Przedstawione metody można zastosować do wielu innych celów. Użyto zapisu właściwego asemblerowi z pominięciem numerów linii. Liczby podane zostały bez dodatkowych oznaczeń w hex. W wielu asemblerach wymagają one poprzedzenia znakiem "$".

```
      *= 4000            Adres startu wybrany dowolnie
FMOVE  = D0B6
FSUB   = DA60
FTEMP  = 0482
FSTOR  = DDA7
FASC   = D8E6
INBUFF = F3
AFP    = D800
CIX    = F2
LBUFF  = 0580
CR     = 9B
PUTREC = 09
GETREC = 05
CIOV   = E456
ICCOM  = 0342
ICBAL  = 0344
ICBLL  = 0348
START  JSR GETNUM        Daje liczbę z E;
       JSR FMOVE         Przenosi ją z FR0 do FR1
       JSR GETNUM        Daje drugą liczbę z E;
       JSR FSUB          FR0 = FR0 - FR1
       BCC NOERR         Przeskok, jeżeli nie ma błędu
       LDA # <ERRMSG     Dane dla IOCB ...
       STA ICBAL         do wyświetlenia komunikatu o błędzie
```

```
       LDA #>ERRMSG
       JMP CONT
NOERR  LDX #<FTEMP        Zapisuje wynik w FTEMP
       LDY #>FTEMP
       JSR FSTOR
                          Następuje konwersja liczby na ciągu zna-
;                         ków, przekształcenie liczby ujemnej na
;                         dodatnią i dodanie znaku nowego wiersza
       JSR FASC           FP na ATASCII, wynik w LBUFF
       LDY #FF
MLOOP  INY
       LDA (INBUFF),Y     Ładuje następny bajt
       BPL MLOOP          Jeżeli dodatni, iść dalej
       AND #7F            Jeżeli nie, kasuje b7 przez maskowanie
       STA (INBUFF),Y
       INY
       LDA #CR
       STA (INBUFF ),Y
;                         Wyświetlanie wyniku
       LDA INBUFF         Podaje adres buforu do IOCB
       STA ICBAL
       LDA INBUFF+1
CONT   STA ICBAL+1
       LDA #PUTREC        Rozkaz "Wpisz rekord"
       STA ICCOM
       LDA #028           Określa długość buforu na 40 dec
       STA ICBLL
       LDA #0
       STA ICBLL+1
       LDX #0             IOCB nr 0 dla edytora ekranu
       JSR CIOV           Wywołanie CIO dla wykonania operacji
       JMP START          w celu ponownego wykonania
                          Odczytanie ciągu znaków cyfr z edytora,
                          zamiana ciągu na liczbę FP, wynik w FR0
GETNUM LDA #GETREC        Daje rekord zakończony znakiem CR
       STA ICCOM
       LDA #<LBUFF        Podaje adres buforu do IOCB
```

```
        STA ICBAL
        LDA #>LBUFF
        STA ICBAL+1
        LDA #28             Określa długość buforu
        STA ICBLL
        LDA #0
        STA ICBLL+1
        LDX #0              IOCB nr 0 dla edytora ekranu
        JSR CIOV            Wywołanie CIO
        LDA #<LBUFF         Wpisuje fo INBUFF adres buforu
        STA INBUFF
        LDA #>LBUFF
        STA INBUFF+1
        LDA #0
        STA CIX             Zeruje wskaźnik buforu
        JSR AFP             Konwersja ciągu ATASCII na liczbę FP
        RTS
ERRMSG  .BYTE "BLAD",CR     .BYTE wprowadza kody liter napisu
       *= 2E0               Informacja dla DOS ...
        .WORD START         o adresie uruchomienie programu
        .END
```

Program umożliwia wprowadzanie dużych liczb i ułamków dziesiętnych, także w tzw. notacji naukowej. Pomija błędnie wprowadzone dane.

Ćwiczenia

1. Ostatni program przyjmuje najpierw odjemnik, a potem odjemną. Co należałoby zrobić, aby odwrócić tę kolejność?

9.10 Wnioski z treści rozdziału

W rozdziale przedstawionych zostało jedynie kilka kwestii związanych z programowaniem w asemblerze na jednym z komputerów wyposażonych w 6502, a mianowicie na Atari. Jednakże nawet ten zwięzły przegląd pozwala lepiej uzmysłowi , jak trudno jest obejść się przy stosowaniu asemblera bez dostatecznej wiedzy o specyficznych cechach komputera, na który piszemy programy.

W języku wysokiego poziomu cechy te są jakby ukryte przed programującym. Część pracy wykonali za niego wcześniej twórcy interpretatora lub kompilatora, wyposażając język w instrukcje i funkcje ułatwiające wykonywanie obliczeń, operowanie grafiką i dźwiękiem, posługiwanie się układami wejścia-wyjścia oraz wiele innych zadań.

Czy jednak można tą drogą wyczerpać wszystkie możliwości komputera? Ich odkrywanie to ciekawe i wdzięczne, choć przeważnie trudne zadanie.

Rozdział 10

ASEMBLERY I PROGRAMY URUCHAMIAJĄCE

10.1 Narzędzia efektywnej pracy

W poprzednich rozdziałach, a w szczególności w pierwszym i ósmym, omówione zostały podstawowe zasady skutecznego tworzenia programów w języku asemblera. Jednakże tak, jak do programowania w Basicu potrzebny jest jego interpretator, tak również warunkiem efektywnego programowania w języku asemblera jest posiadanie przynajmniej trzech podstawowych narzędzi: edytora, który pozwoli napisać program źródłowy, asemblera, który zamieni go w program wynikowy w kodzie maszynowym, oraz debuggera czyli programu uruchamiającego, z którego pomocą potrafimy doprowadzić do usunięcia praktycznie nieuchronnych błędów zapewniając przekształcenie pierwotnego zamysłu w ostateczny kształt dobrego i niezawodnego programu.

Analogia z językiem Basic nie jest pełna. Zawsze pozostaje nam jeszcze jedna droga: napisanie programu w asemblerze na kartkach papieru i wprowadzenie go w kodzie maszynowym. W przypadku mniejszych zadań sposobu tego nie można lekceważyć. Realizacja większych bez asemblera i debuggera jest uciążliwa. Dlatego wyposażenie się w nie jest praktycznie konieczne.

Podstawowe zasady pisania programów w asemblerze 6502 są wspólne dla komputerów wyposażonych w ten mikroprocesor. Natomiast poszczególne asemblery i debuggery mogą się znacznie różnić pod względem skali oferowanych możliwości, jak i składni oraz symbolicznego oznaczenia komend, z których korzysta użytkownik. Dlatego następnym, również pryktycznie niezbędnym, krokiem jest wyposażenie się w instrukcję obsługi programu, najlepiej firmową, w jaką każdy szanujący się wytwórca wyposaża swój wyrób.

W dalszej części rozdziału omówione będą podstawowe zasady korzystania z dwóch programów na Atari: makroasemblera MAC/65 i programu uruchamiającego BUG/65. Opracowała je firma

wytwarzająca oprogramowanie systemowe Optimized Systems Software, której wytworem są m.in. także DOS 2.0S, DOS XL, nowe interpretatory Basic XL i Basic XE, kompilatory języków C i Action! oraz wiele innych programów. Omówienie dotyczy wersji dyskietkowych: MAC/65, wersja 4.20, Stephena D. Lawrowa oraz BUG/65 wersja 2.0 opracowana wspólnie z McStuff Co. Przedstawiony tu opis n i e może zastąpić przewodników dostarczonych przez OSS do obu programów - chociażby dlatego, że razem liczą około 1/3 objętości niniejszej książki. Intencją moją było natomiast dostarczenie, zwłaszcza mniej doświadczonemu programiście, wskazówek ułatwiających posługiwanie się tymi rozbudowanymi i przez to dość skomplikowanymi narzędziami. Podane tu informacje są do tego celu wystarczające.

Zwróćmy uwagę na dwa jeszcze programy.

Pierwszy to "Mastermon" opracowany przez T. Fischermanna w roku 1985 w dwóch wersjach, 2.0 i 2.0a. Pierwsza wprowadzana jest pod adres 8000, a druga 2200 hex, poza tym działają podobnie. Pozwala on pisać programy, ale tylko w JM, ma natomiast wygodne narzędzia do wyświetlania pamięci, a także dezasemblacji bloku pamięci czyli próby jego odczytania jako zapisu w asemblerze.

Program posługuje się jednoliterowymi komendami. Oto niektóre ich przykłady: L - wyświetlanie pamięci w hex, D i U - jej wyświetlanie na dwa sposoby w postaci zdezasemblerowanej, A - zapisywanie w pamięci hex, E - to samo z widocznym poprzednim zapisem. Adresy można podawać w dec i hex (wtedy poprzedzone "$"). Wygodna jest możliwość wyświetlania pamięci małymi fragmentami lub w sposób ciągły - po dodaniu za komendą przecinka i N. Pisząc ?liczba lub ?$liczba otrzymujemy konwersję z dec na hex lub odwrotnie. Pisząc +liczba lub +$liczba powodujemy, że np. D+ lub L+, N spowodują wyświetlanie pamięci od tego adresu. Znak * po komendzie pozwala wznowić wyświetlanie od miejsca, w którym poprzednio zostało przerwane. +* przepisuje pod +adres ostatniego *.

G z adresem dec lub hex uruchamia wykonanie programu użytkownika. Jeżeli chcemy wrócić do Mastermonu, należy na końcu dopisać RTS. Tyle informacji dla tych, którym podobnie jak

autorowi nie udało się dotrzeć do opisu tego programu - z wyglądu całkiem poręcznego. Jego możliwości są bez wątpienia szersze.

W innej sytuacji znajdą się użytkownicy programu EASMD (Editor-Assembler-Debugger) - jednego z wczesnych, lecz nadal użytecznych wytworów firmy Atari. Spotkać można instrukcje napisane do niego po polsku. Ponadto książka D. i K. Inmanów [1] opisuje popularnie pracę na bardzo podobnej, jeszcze wcześniejszej wersji, którą rozpowszechniano na module (kartridżu). EASMD nie ma jeszcze środków tworzenia makrorozkazów, jego debugger nie zapewnia tylu ułatwień, co BUG/65. Korzystne jest natomiast, że w jednym programie połączono pisanie, asemblowanie i uruchamianie.

10.1 Stosowanie makroasemblera MAC/65

Zestaw komend edytora, a także dyrektyw czyli pseudooperatorów oraz operatorów arytmetycznych, logicznych i porównań jest w MAC/65 tak szeroki, że zapewnia skalę możliwości bliską w niektórych dziedzinach oferowanej przez języki wysokiego poziomu. Dochodzą do tego środki makroasemblera, z których pomocą można m.in. wygodnie tworzyć, a potem wykorzystywać biblioteki gotowych fragmentów programów. Przykładem takiej biblioteki jest zestaw makrorozkazów do obsługi wejścia-wyjścia stanowiący plik IOMAC.LIB na dyskietce systemowej i pokrótce omówiony w podręczniku obsługi.

MAC/65 należy do niezbyt szerokiej jeszcze klasy asemblerów, których edytor zapewnia bieżącą kontrolę składniową każdej wprowadzanej linii. Eliminuje to sporą część błędów. Obok takiego trybu sygnalizowanego przez napis-kursor EDIT można zastosować tryb tekstowy, w którym nie ma kontroli składniowej. Osiąga się to pisząc TEXT + RETURN , po czym kursor ma brzmienie TEXTMODE. Jest to tryb umożliwiający pisanie różnych tekstów oraz programów, np. w C. Powrót do trybu EDIT następuje przez napisanie NEW. Przy obu przejściach zerowana jest przestrzeń pamięci edytora. MAC/65 jest kompatybilny "w dół" z EASMD poza drobnymi różnicami.

Omówienie to dotyczy wersji dyskietkowej MAC/65, analo-

giczna dostępna jest na Apple. Opis podzielimy na następujące kwestie: praca w edytorze, środki pomocne w tworzeniu kodu źródłowego, asemblowanie, makrorozkazy.

10.2.1 Co oferuje edytor?

W MAC/65 linie kodu źródłowego w obu trybach muszą być numerowane. Gdy linia nie zaczyna się od numeru, edytor traktuje ją jako linię komend i poprawne komendy wykonuje natychmiast, a niepoprawne pomija sygnalizując błąd napisem: WHAT? (co?).

Szereg komend upraszcza pisanie programu. Np. NUM spowoduje automatyczną numerację linii co 10, NUM 100,5 - rozpoczęcie numerowania od stu z krokiem 5. DEL 100,150 skasuje linie o numerach 100-150. Linię można, podobnie jak w Basicu skasować przez napisanie jej numeru + RETURN . LIST wywoła wyświetlenie listingu, LIST 50,100 - jego części. FIND pozwala znaleźć ciąg znaków w źródłowym listingu. FIND-LDX- pozwoli znaleźć pierwsze wystąpienie LDX, FIND =LDY=,A wskaże wszystkie wystąpienia LDY. Poszukiwany ciąg należy zamknąć w ograniczniki, którymi mogą być dowolne znaki poza cudzysłowem i spacją.

Istotnym uzupełnieniem FIND jest REP (replace) pozwalające zastąpić dawny ciąg znaków nowym. Składnia:

REP /stary/nowy/ [lno1[,lno2]] (,A) (,Q)

Stary i nowy ciąg wpisujemy między ograniczniki, jak przy FIND. Możemy podać numer lub numery linii. Istotne są dwie opcje ostatnie, stosowane z a m i e n n i e. Dopisanie po przecinku A spowoduje, że ciąg nowy będzie zastąpiony przez stary wszędzie, gdzie występuje, natomiast opcja Q sprawi, że przypadki takie będą listowane i przy każdym użytkownik otrzyma pytanie, czy chce dokonać zmiany.

Choć MAC nie jest debuggerem, włączono do niego dwie komendy umożliwiające dotarcie do kodu maszynowego w dowolnym miejscu pamięci. D 100,200 wyświetli w hex zawartość bajtów od 100 do 200 hex. Komenda C umożliwia zmiany w pamięci. Jej składnia jest następująca:

C hxnum1< (,) hxnum [(,) (,hxnum) ...]

hxnum1 oznacza adres początku zmienianego programu w hex, dalsze dane - wartości wprowadzane w hex do kolejnych bajtów o dowolnej liczbie. Dodatkowy przecinek powoduje opuszczenie komórki.

Na przykład: C 4000<72,00,A8,,5

Spowoduje to zmianę zawartości komórek: 4000 na 72, 4001 na 00, 4002 na A8, 4003 będzie opuszczone, a 4004 zmienione na 5.

Edytor zawiera szereg komend dla współpracy ze stacją dyskietek. BLOAD i BSAVE pozwalają analogicznie jak komendy DOS wprowadzać i wyprowadzać pliki binarne. SAVE i LOAD podobnie jak w Basicu zapisują na dyskietkę kod źródłowy z przestrzeni edytora lub wpisują go do niej.

LIST#D:TEST zapisze listing do pliku TEST na dyskietce, natomiast PRINT #D:SOURCE uczyni to samo, ale listing będzie zapisany do SOURCE bez numerów linii. Do ponownego wprowadzenia tak zapisanych plików służy ENTER, przy czym ENTER,M przyłączy tekst do znajdującego się już w przestrzeni edytora, a ENTER,A spowoduje ponumerowanie linii wprowadzanego tekstu, konieczne np. po wpisaniu na dyskietkę z pomocą PRINT. A zawsze oczyszcza przestrzeń edytora.

Komenda DOS zapewnia powrót do DOS, a BYE - powrót do testu pamięci Atari.

10.2.2 Pisanie programu źródłowego.

Budowę poprawnej linii kodu źródłowego omówiliśmy już w punkcie 1.7 i stosujemy w książce. W MAC/65 zasady są takie same. Linia kodu źródłowego musi zaczynać się od jej numeru, po którym obowiązkowa jest jedna spacja. W następnej kolumnie musi zaczynać się etykieta. Mnemonik asemblera wpisujemy najwcześniej w trzeciej kolumnie (można dalej) za numerem linii lub po co najmniej jednej spacji za etykietą. Operand może zaczynać się gdziekolwiek za mnemonikiem kodu operacji, a komentarz - gdziekolwiek za operandem lub, gdy go nie ma, mnemonikiem. Jak widać, ściśły przepis co do miejsca rozpoczęcia dotyczy tylko etykiety, co można wykorzystać do przejrzys-

tego formatowania tekstu źródłowego.

Brzmienie mnemoników jest takie, z jakiego korzystamy w książce. Etykieta w MAC/65 musi zaczynać się od litery, znaku @ lub znaku zapytania, można w niej użyć liter, cyfr i kropki. Prawdziwie imponująca jest dopuszczalna długość etykiety - aż 127 znaków i wszystkie znaczące. Możliwa jest zatem etykieta:

OD.TEGO.MIEJSCA.JEST.PROGRAM.OBSŁUGI.IOCB.NR.0

Mniej utrudzimy asembler, gdy napiszemy to w komentarzu. Jeżeli w linii jest tylko komentarz, po numerze linii oraz spacji wpisujemy średnik lub gwiazdkę. Przy asemblowaniu linia taka będzie pominięta.

Poza tym standardowym sposobem pisania programu źródłowego MAC/65 pozwala na wpisywanie do linii programu źródłowego dwóch rodzajów oznaczeń: operatorów i dyrektyw.

Operatory - to dobrze znane z języków wysokiego poziomu symboliczne nazwy operacji pozwalające budować w y r a ż e n i a. Gdy MAC napotka w linii takie wyrażenie, obliczy je i jedynie wynik wykorzysta w zasemblowanym kodzie.

Jeżeli np. zdefiniujemy etykiety P jako 1024, a R jako 7, to można napisać: LDA P*R+2. Spowoduje to, że rozkaz uzyska postać LDA 7170 i tak zostanie zapisany w kodzie wynikowym. W MAC można stosować operatory czterech działań arytmetycznych, wszystkich sześć operatorów relacji zapisywanych identycznie jak w Basicu (=,< ,>= itd) oraz operatory logiczne działające podobnie jak w Basicu i zapisywane z kropką na początku: .AND, .OR, .NOT. MAC ma ponadto bitowe operatory logiczne, których znakami są: kreska pionowa (OR), & (AND) i znak potęgowania (EOR).

W przypadku, gdy np. w Basicu musielibyśmy zastosować w obliczeniach nawiasy, w MAC trzeba użyć nawiasów kwadratowych, ponieważ okrągłe wykorzystuje się w zapisie trybów adresowania pośredniego.

Z użytecznych operatorów unarnych (tzn. mających jeden argument) korzystaliśmy: <IRQ oznacza młodszy bajt adresu zaznaczonego etykietą IRQ, a >IRQ - starszy bajt tego adresu.

W asemblerach spotkać można również inne oznaczenia. Unarny minus zmienia znak liczby. Istnieją ponadto operatory .DEF i .REF, które tu pominiemy.

D y r e k t y w y - to inny ważny instrument upraszczający pisanie programów. Z ich stosowaniem łączy się tyle odcieni możliwości i typów zadań, że ograniczymy się do zwięzłego omówienia najczęściej stosowanych spośród ogółem 21 dyrektyw.

Trzy dyrektywy łączą się z różnymi sposobami przypisywania wartości licznikowi programu i etykietom. W niektórych z nich występuje specjalne oznaczenie: gwiazdka "*", która w takim wypadku oznacza bieżącą wartość licznika programu. Połączenie znaków "*=", pisanych bez odstępu między nimi stanowi dyrektywę przypisującą licznikowi programu wartość napisanego za nią adresu. Inne znaczenie ma kolejna dyrektywa: znak równości. Służy ona do przypisania na stałe wartości etykiecie. Pisząc na przykład:

```
DOSINI = $C
```

powodujemy, że odtąd w całym programie etykieta DOSINI będzie oznaczała taką wartość.

Odmienną rolę pełni trzecia z pokrewnych dyrektyw ".=" (kropka i znak równości). Etykieta zdefiniowana z jej pomocą może potem otrzymać inną wartość, co nie jest możliwe przy "=".

Drugą ważną grupę stanowią dyrektywy umożliwiające wpisywanie w rozmaitym trybie do programu źródłowego pojedyńczych wartości lub całych ich ciągów.

.BYTE pozwala wprowadzać do kodu wynikowego wartości pojedyńczych bajtów. Na przykład fragment

```
110 .BYTE "ABC",3,-1
```

spowoduje wyprowadzenie w kodzie wynikowym wartości:

```
41 42 43 02 FF
```

.SBYTE powoduje również wyprowadzenie pojedyńczych bajtów, ale w kodzie wewnętrznym stosowanym np. przez ANTIC w obsłudze ekranu. Na przykład

120 .SBYTE "Hallo!"

spowoduje na Atari wyprowadzenie następujących kodów wewnętrznych:

28 61 6C 6C 6F 01

.CBYTE działa identycznie jak .BYTE, a jedynie w kodzie ASCII ostatniego znaku dokonuje inwersji najwyższego bitu. Jest to często stosowana metoda prostego zaznaczenia końca łańcucha znakowego. Inną metodą jest kończenie łańcucha jedną lub kilku wartościami 0. Przykład:

200 .CBYTE 1,"SYSTEM"

wytworzy to ciąg: 01 53 59 53 54 45 CE. Kod znaku "M" został zwiększony o 80 hex.

.DBYTE i .WORD służą do umieszczenia w kodzie wynikowym jednego lub więcej słów dwubajtowych. Różnica polega na tym, że .DBYTE nie zmienia przy tym kolejności bajtów, a .WORD przestawia bajt młodszy ze starszym. Ta druga dyrektywa jest bardziej użyteczna, bowiem zgodna jest ze sposobem interpretowania przez 6502 adresów. Przykład:

```
.DBYTE $1234,1,-1      wytworzy: 12 34 00 01 FF FF
.WORD $1234,1,-1       wytworzy: 34 12 01 00 FF FF
```

Istotną cechą opisywanego makroasemblera jest to, że umożliwia tzw. asemblowanie warunkowe. Służy do tego zestaw dyrektyw: .IF, .ELSE, .ENDIF.

Napotkanie .IF powoduje, że wyrażenie bezpośrednio po nim następujące jest obliczane. Jeżeli daje wartość niezerową, do kodu wynikowego wprowadzany jest następny fragment kodu aż do napotkania .ELSE lub .ENDIF. Jeżeli wyrażenie ma wartość 0, fragment, o którym mowa, jest pomijany w kodzie wynikowym i asemblowanie wznowione zostaje dopiero za .ELSE lub .ENDIF. Rolę asemblowania warunkowego rozpatrzymy dalej na przykładzie makrorozkazu.

Wygodna dyrektywa .INCLUDE pozwala włączyć do programu inny program ź r ó d ł o w y. Na przykład:

.INCLUDE D:SYSEQU.M65

Włączy to do kodu źródłowego plik systemowy etykiet związanych z obsługą wejścia-wyjścia na Atari opracowany przez OSS i oszczędzi sporo pracy, gdy chcemy programować takie zadanie.

10.2.3 Asemblowanie

Po opracowaniu programu źródłowego chcemy go z reguły przekształcić w kod wynikowy czyli zasemblować. Korzystamy przy tym z komendy edytora ASM. Przy asemblowaniu pomocne są także komendy SIZE i LOMEM oraz dyrektywa .OPT. Omówmy najpierw sposób korzystania z trzech ostatnich rodzajów zleceń. MAC/65 umożliwia asemblowanie zarówno na dyskietkę, jak i do pamięci. Gdy jednak chcemy, by kod wynikowy znalazł się w pamięci, wymaga to pewnych środków ostrożności: trzeba zapobiec temu, by nie wszedł na miejsce zajęte przez program źródłowy czy też sam MAC/65, który zajmuje adresy powyżej 8800 hex. Gdy wybieramy dla naszego programu te wysokie adresy, nie ma innego sposobu niż asemblowanie na dyskietkę.

Ważną cechą MAC/65 jest to, że bez wyraźnego zlecenia nie będzie nigdy asemblować do pamięci. Zleceniem tym jest dyrektywa .OPT OBJ, którą musimy w p i s a ć na początek programu źródłowego. Dyrektywa .OPT ma także szereg innych zastosowań, których nie omówimy, np. .OPT NO LIST spowoduje, że nie będzie wyprowadzany listing asemblacji. .OPT OBJ oznacza, że chcemy asemblować do pamięci. Gdy brak tej dyrektywy, MAC przez domniemanie przyjmuje .OPT NO OBJ.

Stosując .OPT OBJ trzeba dokładnie upewnić się, że jest to bezpieczne. Np. małe programy możemy zawsze asemblować na stronę 6, ponieważ MAC/65 zostawia ją wolną. W pozostałych przypadkach użyteczna jest komenda edytora SIZE. MAC podaje po niej trzy liczby: pierwsza jest najniższym adresem dostępnej pamięci, druga wyznacza górną granicę obszaru zajętego przez nasz program źródłowy, trzecia - szczyt pamięci. Można asemblować tylko do obszaru między drugim, a trzecim adresem. Nie niżej i nie wyżej. Jednakże dolną granicę można zmienić z pomocą komendy LOMEM. Pisząc na przykład:

```
LOMEM 6000
```

powodujemy, że program źródłowy będzie umieszczany powyżej 6000 hex. Należy to, oczywiście, uczynić p r z e d pisaniem lub wprowadzeniem do pamięci programu źródłowego. Wtedy adresy powyżej obszarów zajętych przez OS i DOS do wysokości 6000 hex będą bezpieczne dla kodu wynikowego.

Asemblowanie do pamięci ma zatem szereg wad, ale i pewną zaletę. Po zasemblowaniu naszego programu możemy wrócić do DOS, wywołać BUG/65 i program od razu "szlifować". Przedtem jednak musimy wykorzystać możliwości usunięcia błędów, jakie zapewnia MAC/65 w czasie asemblowanie.

Wykonanie asemblowania wywołuje komenda ASM. Jeżeli w programie nie napisaliśmy .OPT OBJ, to po napisaniu:

ASM

uzyskamy to, że nasz program będzie asemblowany, na ekran wyprowadzone zostaną komunikaty o błędach, ale kod wynikowy ... nie będzie wytwarzany. Oto zaleta tego mechanizmu: możemy usunąć wszystkie dostrzeżone przez MAC błędy i dopiero potem wykonać asemblację.

ASM powoduje asemblowanie pliku źródłowego MAC/65, wyprowadza listing i kod wynikowy. Składnia komendy jest następująca:

ASM #file1, #file2, #file3, #file4

file1 oznacza urządzenie (plik z dyskietki lub pamięć) zawierające kod źródłowy, file 2 - urządzenie dla wyprowadzenia listingu, file3 - dla kodu wynikowego, natomiast file4 oznacza plik tymczasowy tworzony przez MAC/65.

Jeżeli nie określi się nazw plików (zastępuje się wtedy je przecinkami) MAC przyjmie przez domniemanie, że są to:

file1 - kod źródłowy użytkownika w pamięci
file2 - edytor ekranu
file3 - pamięć (tylko gdy poda się .OPT OBJ)
file4 - nie ma.

Oto przykłady stosowania komendy:

ASM #D:SOURCE , , #D:OBJECT

Kod źródłowy będzie czytany z pliku o nazwie SOURCE, a

wynikowy wpisany do pliku OBJECT. Listing wypisany będzie na ekranie:

```
ASM ,#P: , ,#D:TEMP
```

Kod źródłowy będzie czytany z pamięci, listing wyprowadzony na drukarkę, kod wynikowy zapisany w pamięci, a plik tymczasowy na dyskietkę.

10.2.4 Makrorozkazy

Makrorozkazy - to fragmenty kodu tworzone i wykorzystywane w specjalny sposób. MAC/65 wprowadza zdefiniowane rozkazy do odrębnego obszaru pamięci, a podczas asemblowania włącza je do kodu źródłowego wykonując przy tym dodatkowy trzeci przebieg. Stosowanie makrorozkazów składa się z dwóch odrębnych faz: ich definiowania oraz ich wykorzystania czyli, jak to się określa, r o z s z e r z a n i a. Ważną cechą makrorozkazów jest możliwość przekazywania do nich parametrów z zasadniczego programu.

Do definiowania nowych makrorozkazów służy para dyrektyw:

```
.MACRO nazwa-makrorozkazu
.ENDM
```

Między nimi, jak w klamrze umieszcza się kod.

Gdy zdefiniujemy makrorozkaz, jego nazwa może być wykorzystana w programie źródłowym, raz lub wielokrotnie, zamiast mnemoniku rozkazu. Rozpatrzmy prosty przykład makrorozkazu wykonującego przechowanie na stosie zawartości rejestrów A, X i Y.

```
.MACRO PRZECHOWAJ
PHA
TXA
PHA
TYA
PHA
.ENDM
```

Po zdefiniowaniu tego makrorozkazu możemy go teraz włączać do kodu źródłowego analogicznie jak mnemoniki.

Z pomocą makrorozkazów można budować biblioteki fragmentów

programów o dowolnej długości i następnie włączać je do pisanych programów. Np. twórcy MAC/65 opracowali zbiór makrorozkazów do obsługi wejścia-wyjścia przedstawiony w pliku IOMAC.LIB i skomentowany w instrukcji. Makrorozkazy zbudowano w ten sposób, że najprost^ze są wykorzystywane we wnętrzu bardziej złożonych. Ponadto posłużono się zbiorem etykiet zdefiniowanych w SYSEQU.

Do makrorozkazów można przekazywać p a r a m e t r y. W samym wnętrzu makrorozkazu parametry oznaczane są jednolicie numerami poprzedzonymi znakiem % dla wyrażeń liczbowych lub parą znaków %$ dla łańcuchów znakowych. Rozpatrzmy makrorozkaz @CH, wykonujący ładowanie do rejestru X numeru IOCB pomnożonego przez 16. Przyjmuje się, że gdy podana zostanie wartość od 0 do 7, oznacazać będzie numer kanału, natomiast wartość większa oznaczać będzie adres, pod którym znajduje się numer kanału.

```
.MACRO @CH
.IF %1>7
LDA %1              Numer IOCB podano w pamięci
ASL A
ASL A
ASL A
ASL A               Numer IOCB pomnożony przez 16
TAX                 i przeniesiony do rejestru X
.ELSE
LDX#%1*16           Numer podano wprost
.ENDIF
.ENDM
```

W definicji tego makrorozkazu poznajemy także zastosowanie asemblowania warunkowego. Zależnie od wielkości parametru zostanie on potraktowany odmiennie i za każdym razem inna część makrorozkazu zostanie zasemblowana.

Jaka jest forma stosowania tego makro? W tekście źródłowym należy podać jego nazwę, a za nią wypisać parametry, w tym wypadku jeden - numer IOCB lub adres, pod którym numer ten się znajduje. Oczywiście, możliwe jest przekazywanie w ten sposób wielu parametrów - do 63 w jednym makro oznaczanych np. %7,

%Ø22, %63. Specjalne znaczenie mają symbole %O i %ØO. Podają one mianowicie liczbę parametrów zastosowanych w danym makrorozkazie.

10.3 Program uruchamiający BUG/65

BUG/65 oferuje szerokie możliwości sprawnego uruchamiania i poprawiania programów, o czym była już mowa w rozdz. 1. Zapewnia pełny zestaw komend pozwalających wyświetlać i zmieniać pamięć oraz rejestry 6502, uruchamiać program użytkownika z ustanowieniem punktów przerwania. Z pomocą BUG można czytać z dyskietki i wpisywać na nią programy z ich ewentualnym dołączeniem (append) do już wpisanych. Stosując przesunięcie adresu startu można umieścić badany program w innym niż przewidziany dla niego obszarze pamięci. BUG umożliwia także odczyt i zapis treści dyskietki sektorami.

Podręczny asembler BUG/65 daje poza omówionym stosowaniem 10 etykiet także możliwości analogiczne do .BYTE, czemu służy pseudooperator "+". Pisząc np. +34 pod aktualną wartość licznika pamięci wprowadzamy 34 hex. Pseudooperator "/" pozwala ustanowić nowy stan licznika programu PC.

BUG/65 umożliwia dezasemblację programu czyli odczytywanie z pomocą komendy Y jego budowy. Można w nim stosować liczby hex i dec, te ostatnie zaznaczane z pomocą kropki, używać operatorów + i - .

Dwie istotne cechy BUG/65 - to możliwość stworzenia jego nieprzemieszczalnej wersji lokującej go w wybranym przez nas obszarze pamięci. Standardowo umieszczany on jest bezpośrednio nad dołem wolnej pamięci LOMEM. Druga cecha - to ochrona obszaru BUG/65 przed komendami, które spowodowałyby wkroczenie na ten obszar i ewentualne zniszczenia.

BUG/65 tworzy dla potrzeb użytkownika kopię strony zerowej, którą sam, oczywiście, wykorzystuje. Możemy zatem zbadać programy mając wrażenie, że pracują one na stronie zerowej. Jednakże uruchomienie takich programów z pomocą U może zakończyć się niepowodzeniem.

Istotnym elementem doskonalenia własnych umiejętności

jest analiza programów napisanych przez innych. Można je, niestety, zbyt rzadko, znaleźć w czasopismach krajowych lub zagranicznych. Znacznie szerszym źródłem takich programów są ... nasze własne dyskietki i kasety magnetofonowe. Odczytanie ich z pomocą debuggera pozwala na powrót przeobrazić niezrozumiałe ciągi liczb w zapis w asemblerze. Program uruchamiający nie potrafi nam na ogół dać zupełnie wiernego odtworzenia konstrukcji programu, ponieważ jego tekst jest często przemieszany z fragmentami danych, w tym kodów znaków do wyświetlenia. W tym ostatnim wypadku pomocne może być to, że programy uruchamiające wyświetlają po prawej stronie ekranu znaki, które w kodzie ASCII odpowiadają kolejnym liczbom.

10.3.1 Śledzenie wykonania programów

Jeżeli program zawiera błędy i chcemy je z niego usunąć, BUG/65 oferuje zestaw ułatwiających to komend. Wstępną czynnością jest przejrzenie programu z pomocą omówionej już komendy Y. Pomocna jest również komenda V wyświetlająca zawartość wszystkich rejestrów, przy czym w rejestrze P podane są wartości znaczników pod ich nazwami.

Dalszą użyteczną czynnością jest wykonanie tzw. śledzenia (ang. trace) programu. Z pomocą komendy X powinniśmy do rejestru PC wpisać adres startu naszego programu umieszczonego w pamięci. Gdy napiszemy: XP, pojawi się napis "P=" i tu wpisujemy nasz adres. Komenda wyświetla kolejno rejestry pozwalając zmieniać ich zawartość. RETURN lub ESC kończy te czynności.

Do śledzenia służy komenda T. Pisząc np. T5 powodujemy, że zostanie wykonanych i wyświetlonych na ekranie pięć kolejnych rozkazów asemblera. Podane będą stany rejestrów po wykonaniu każdego rozkazu. Komenda TS umożliwia wykonanie tego samego, jednak z pominięciem podprogramów. Pozwala to obejrzeć sam program główny.

Należy podkreślić, że rejestr P, który widzimy, jest jedynie pseudorejestrem prowadzonym przez BUG/65. Mimo to w toku śledzenia wykonywane są czynności przewidziane w naszym

programie. Komplikacje mogą powstać, gdy program korzysta z tych części strony zerowej, które wykorzystuje również BUG/65.

Skróconą formę śledzenia programu zapewnia komenda G. Jej stosowanie wymaga dużej dokładności. Składnia jest następująca:

G start [@brkpoint [RN=wartość [I=razy]]]

Nawiasy kwadratowe oznaczają, że ujęte w nie fragmenty można opuścić. start - to adres uruchomienia naszego programu. @brkpoint - to poprzedzony znakiem "@" adres miejsca w programie, w którym chcemy zatrzymać jego wykonywanie. W ten sposób możemy sprawdzić, czy program do tego miejsca działa. Nosi to nazwę ustanawiania punktu przerwania (ang. breakpoint).

Końcowe dwa opcjonalne fragmenty linii zapewniają dodatkowe możliwości próbnego wykonania naszego programu. Zdarza się, że np. przy wielokrotnym wykonywaniu przez program pętli chcielibyśmy go zatrzymać dopiero wtedy, gdy, powiedzmy, rejestr X przybierze wartość 0 lub FF. Piszemy wówczas: RX=0 lub RX=FF. Możemy także życzyć sobie zatrzymania po np. 100-krotnym wykonaniu pętli. Wpisujemy wtedy: I=64 (64 - to w hex odpowiednik 100). Oto przykłady zastosowania tego mechanizmu z wyjaśnieniami:

G 2000 idź pod adres 2000 i wykonaj program bez jego przerywania

G @4000 idź z miejsca wskazanego przez Twój licznik programu i przerwij, gdy dojdzie on do 4000

G 1000 @1411 RY=35
idź od adresu 1000 i przerwij w miejscu 1411 tylko wtedy, gdy w rejestrze Y będzie wartość 35

G 1000 @1414 I=2
idź od 1000 i przerwij pod 1414, gdy Twój program po raz drugi znajdzie się w tym punkcie.

Tak więc G umożliwia sprawdzenie naszego programu niejako dużymi skokami. Natomiast T i TS pozwalają na drobiazgową analizę przebiegu realizacji programu. Można wówczas wykryć w

nim na przykład, że zastosowaliśmy nieodpowiedni rozkaz odgałęzienia, że liczba pętli nie jest taka jak zakładaliśmy, że rejestr nie przybiera oczekiwanej przez nas wartości, że znacznik C nie zmienił się po DEX, a mylnie tego oczekiwaliśmy itd.

10.3.2 Przegląd komend

A oto skrótowe omówienie niektórych pozostałych komend BUG/65. W nawiasach kwadratowych - nie obowiązkowe fragmenty linii komendy.

A adres (naciśnięcie spacji)

Działa podobnie jak S, ale pozwala wprowadzać do komórek pamięci kody ATASCII znaków, które napiszemy. By wpisać znak sterujący, należy go poprzedzić znakiem "@". Wydrukowanie znaku podkreślenia "_" umożliwia cofnięcie się we wprowadzaniu do poprzedniej komórki. To samo odnosi się do komendy S. Klawisz ESC powoduje powrót do trybu wprowadzania komend.

C startblok1 endblok1 startblok2

Umożliwia porównanie dwóch bloków pamięci blok1 i blok2.

F start end [wartość]

Wypełnia blok od start do end wartością. Jeżeli nie podamy wartości, wypełni blok zerami.

H liczba1 liczba2

Wyświetla w następnym wierszu sumę i różnicę dwóch liczb hex.

I

Powoduje wyświetlenie katalogu dyskietki.

K liczba hex

Wyświetla wartość dziesiętną liczby hex. By wykonać odwrotną konwersję, można zastosować komendę D podając liczbę dziesiętną poprzedzoną kropką, np. D .256. Wyświetlony zostanie numer linii 0100 - odpowiednik 256 w hex.

L start end bajt1 bajt2 ... bajtN

Lokalizuje łańcuch bajtów w obszarze pamięci od start do end. Na przykład: L 1000 10FF 41 42 43 - pozwoli ustalić, czy

między adresami 1000 a 10FF znajduje się jeden lub więcej łańcuchów "ABC". Wszystkie adresy ich występowania zostaną podane.

M sstart end dokąd

Przesuwa blok od start do end w obszar od adresu dokąd.

P [S][P]

Służy do wyprowadzenia na ekran (S) lub drukarkę (P) albo też na oba te urządzenia (SP) listingów tworzony h przez komendy BUG/65.

Q

Przejście do DOS.

R przesunięcie #D:nazwa

Ładuje plik binarny z dyskietki pod wskazany w nim adres. Jeżeli użyjemy "przesunięcia" dodatniego czy ujemnego, treść pliku umieszczona zostanie wyżej lub niżej o odpowiednią liczbę bajtów niż adres podany w programie.

R% nr - sektora adres liczba - sektorów

Pozwala wgrać do pamięci pod adres z dyskietki: nr - sektora oznacza numer pierwszego wpisywanego sektora, a liczba-sektorów - ich liczbę. Gdy jej nie podamy wgrany będzie jeden sektor.

W [:A] start end #D:nazwa

Pozwala zapisać na dyskietkę fragment pamięci od start do end w pliku "nazwa". Jeżeli po komendzie dodamy ":A", nastąpi dołączenie wpisywanego fragmentu do istniejącego pliku.

W% numer-sektora adres-buforu liczba-sektorów

Pierwsza liczba - to numer sektora, w którym ma być rozpoczęte wpisywanie, druga - adres początku wpisywanego obszaru pamięci, a trzecia - liczba sektorów, które mają być zapisane. Jeżeli jej nie podamy, zapisamy będzie jeden sektor.

10.4 Przenoszenie programów na DOS 2.5

MAC/65 i BUG/65 opracowane zostały do współpracy z DOS XL. Może to być niewygodne dla posiadaczy Atari 130XL lub 800XL z rozszerzoną pamięcią, ponieważ nie pozwala korzystać z bardzo

wygodnej przy pracy nad programami dyskietki wewnętrznej.

Jednakże oba programy można łatwo przystosować do współpracy z DOS 2.5. Sprawa polega na tym, że DOS XL tworzy nieco inny mechanizm zapisywania adresu startu programów, co powoduje, że programy po wgraniu do pamięci z DOS 2.5 często nie startują automatycznie.

MAC/65 można uruchamiać stosując w DOS 2.5 opcję M i podając adres startu 8800 hex. Najpewniej jednak będzie wgrać MAC/65 z DOS 2.5, po czym zapisać go ponownie na dyskietkę opcją K (SAVE) podając kolejne adresy:

MAC65, 8800, BA3B, 8800

Tak przygotowana wersja będzie doskonale wpółpracować z DOS 2.5 i dyskietką wewnętrzną.

Nieco kłopotliwsze jest dostosowanie BUG/65, którego zasadnicza wersja jest przemieszczalna. Jednakże program ten można również przygotować z DOS XL w wersji nieprzemieszczalnej. W tym celu po wyjściu z menu DOS XL do linii komend piszemy w niej po nazwie adres. Powiedzmy, że chcemy przygotować wersję ulokowaną bezpośrednio pod MAC.BUG/65 ma długość 1E00 hex. Z DOS XL wgrywamy BUG następująco:

D1: BUG65 6A00

Tak stworzonę wersję nieprzemieszczalnę zapisujemy na dyskietkę:

D1: SAVE BUG6A00 6C00 87FF

Wersja ta będzie pracować pod DOS.2.5 po starcie opcją M pod adresem 6C00. Najwygodniej jednak będzie wgrać ją z DOS 2.5 opcją K podając adresy 6C00, 87FF, 6C00. Prawdopodobnie nie będzie działać w sposób właściwy opcja Q - powrót do DOS. Wymaga to korekty podprogramu wywoływanego tą opcją, który w danym wypadku znajdzie się pod adresem 7C4B. Trzeba wcześniej znieść ochronę pamięci, którą nakłada na swój obszar BUG. W tym celu pod wolnym adresem w pamięci, np. 5000 hex, wpisujemy opcją Z program:

```
LDA #0
STA 6C1E
RTS
```

Po U 5000 program jest odbezpieczony. Opcją Z lub S pod adresem 7C4B wpisujemy:

```
JMP (000A)              6C 0A 00
```

Przywracamy ochronę wpisując do 6C1E wartość niezerową. Zapisujemy BUG6C00 ponownie na dyskietkę. Wersja ta będzie działać także pod Sparta DOS-em. Można, oczywiście, wybrać inny adres początku BUG w pamięci.

Opisana tu metoda może być użyteczna w przypadku innych programów zapisanych na dyskietkach z pomocą DOS XL. Jest on dość zachłanny i wpisuje nawet na dyskietki swoje przeróbki. Ostatnio na Atari pojawiają się nowe DOS, np. Sparta DOS i TOP DOS. Wykorzystuje się w nich postęp osięgany w budowie nowoczesnych dyskowych systemów operacyjnych na większe mikrokomputery, takich jak CP/M czy UNIX. Należy sądzić, że komputery 8-bitowe oparte na 6502 będą nadal owocnie wykorzystywane.

x

x x

Nasz przewodnik po asemblerze i języku maszynowym mikroprocesora 6502 dobiegł końca. Jeżeli pobudzi dalszy rozwój zainteresowań Czytelników tymi użytecznymi i ciekawymi językami, główne zadanie tej książki można będzie uznać za spełnione.

ROZWIĄZANIA ĆWICZEŃ

Podano rozwiązania ćwiczeń oznaczonych "x" w układzie: punkt książki, numer ćwiczenia i rozwiązanie.

1.1.1 - 3. Etykieta nie może być oznaczeniem kodu operacji rozkazu.

1.5 - 1. 255, 65, 192, 79.

1.5 - 3. d/ 110111011100000.

1.6 - 1. Algorytm opisuje słownie lub graficznie sposób wykonania zadania, natomiast program wyraża to w jednym z języków programowania umożliwiając realizację zadania.

1.7.1 - 1. d/ 100 hex nie pomieści się w 8-bitowym akumulatorze. f/ STA nie ma trybu natychmiastowego, nie można bowiem zapisać wartości w ... wartości, a jedynie w pamięci.

1.8 - 1:

```
a/ LDA G3        d/ LDA =0        INC E4
   STA G5           STA Z         LDA E4
                                  STA E6
```

1.8 - 2. b/ U=62:W=62 c/ T=S:T+1:U=S

1.8 - 3. a/ 2 b/ 3 c/ 1 d/ rozkaz zapisany błędnie; INC wymaga operandu e/ 1.

1.9 - 1. A=39:L1=A:L2=A:L2=L2+1

1.9 - 2:

```
LDA #7F
STA M
STA K
STA L
INC L
INCL
```

2.5 - 1. A, X, Y, P, PC (PCL i PCH), S, IR.

2.6 - 1. FF hex czyli 255 dec.

2.7 - 1. Szczyt stosu po prawej:

Stos pusty; 27; 27 27; 27 27; 27,27,30; 27,27; 27; 27,27; 27.

2.7 - 2. Nie, ponieważ nie zdjęliśmy wszystkich wstawionych wartości przed zakończeniem programu.

3.5 - 3:

```
  110111          10110          100101
- 100100        - 1011         -    111
  010011          1011           11110
```

3.5 - 4:

```
CLD
LDA K
CLC
ADC L
STA M
LDA K+1
ADC L+1
STA M+1
```

3.5 - 5. Nie, jeżeli dodanie starszych bajtów ustawi znacznik C, co będzie oznaczało, że suma nie zmieściła się na 16 bitach.

3.7 - 1. a/ Suma poprawna b/ i c/ - niepoprawne.

3.8 - 1:

```
b/ LDA P          d/ LDA W
   SEC               SEC
   SBC Q             ADC #9
   STA Q             SBC T
   INCQ              STA W
```

Wykonanie INC po odejmowaniu zapobiega błędowi, gdy P=FF hex. W drugim przykładzie ustawienie znacznika C przed dodawaniem da poprawny wynik, gdy W+9 nie jest większe niż FF, nastąpi bowiem wówczas w rzeczywistości W+(9+1) - (T+1).

3.8 - 3. M+5 oznacza a d r e s o 5 wyższy niż M.

3.9 - 2. Jest to program rozpatrywany przed chwilą.

3.10 - 2. Jest to dzielenie całkowitoliczbowe i wynik jest zaokrąglony w dół o 0.5.

3.11 - 4. 90; 144. W kodzie BCD 4-krotne ASL jest równoznaczne z mnożeniem przez 10, a 4-krotne LSR - z dzieleniem przez 10.

4.2 - 1:

```
LDA 1000
PHA
LDA 2000
PHA
```

4.2.2 - 1. b/:

```
     11110110
ORA  10100111
     --------
     11110111
```

4.4.3 - 1. Wyniki: a/ 15 b/ 31 c/ FF d/ 82.

4.6 - 1.

a/
```
LDA F
CLC
ADC G
CMP H
BEQ L20
```

b/
```
INC T
LDA T
CMP U
BNE L20
```

4.6.1 - 1. Jest niepoprawna. Przy każdej wartości C wykonany będzie następny rozkaz.

4.6.2 - 1. Można, ale pętla wykonana będzie o jeden raz więcej.

5.4 - 1. FOR I=1 TO 224: POKE 3000+I, PEEK (2000+I) :NEXT I

5.8 - 1:

```
     LDY #3F
CYKL CLC
     LDA 3000,Y
     ADC 3040,Y
     STA 3080,Y
     BPL CYKL
```

6.3 - 1:

a/
```
    1010
 1000110:111
  111
  ---
    111
    111
    ---
     00
```

b/
```
     110
 1000000:1010
  1010
  ----
   1100
   1010
   ----
    100
```

c/
```
      1111
11001000:1101
1101
 11000
  1101
  10110
   1101
   10010
    1101
     101
```

6.6 - 1. b/:

```
LDX #6
LDA K
STA T,K
```

6.6 - 2. a/ U(J)=T(J) b/ U(M)= U(M)-1

6.9 - 1. Zmodyfikowany wariant jest szybszy i krótszy, zajmuje jednak rejestr Y, który w pierwszym wariancie jest wolny.

6.9 - 2. Przesuwa bity w Q o 7-X w prawo.

6.11 - 1. Linie, w których użyto adresu TEMP, należy zmienić następująco:

```
STA TEMP+1        STA GOR+1
STA TEMP          STA DOL+1
LDA TEMP          DOL LDA #0
LDA TEMP+1        GOR LDA #0
```

7.5 - 2. Jedna z możliwości. Linie 60-80 należy skasować i pisac:

```
60 X=USR(adres,D)
```

gdzie "adres" oznacza początek następującego programu:

```
       PLA          Zdejmuje liczbę parametrów
       PLA          Zdejmuje starszy  zerowy  bajt D
       PLA          Kod ASCII znaku
       BEQ KROPKA   Jeżeli kod=0, skok pod KROPKA
       CMP #9B      Czy kod RETURN — 155 dec?
       BNE DRUK     Jeżeli nie, drukować
KROPKA LDA #2E      Kod kropki w A
DRUK   JSR F2B0     Podprogram druku znaku
       RTS          Powrót do Basicu
```

8.2 - 1. Na przykład:

```
    JMP L4
L3  CLC
    ADC #5
    STA K
L4  LDA K
    CMP L
    BCC L3
```

8.2 - 2. a/:

```
10 IF WAR THEN 1000
20 SEKW:GOTO 2000
1000 Q
1010 IF NOT D THEN 1000
2000 END
```

8.3 - 3. Za pierwszym razem ładowany jest zerowy, a za drugim ostatni element tablicy.

BIBLIOGRAFIA

1. Inman Don, Inman Kurt: The Atari Assembler. Reston, USA, 1981, (s. 270)

2. Mansfield Richard: Second Book of Machine Language. COMPUTE! Books, 1985, s. 459

3. Morer (Maurer) U: Jazyk assemblera dla piersonalnogo kompiutiera EPŁ. Moskwa 1987, Mir. Przekład książki: W. Douglas Maurer: Apple Assembly Language. A course of study based on LazerWare software. Computer Science Press, 1984, s. 430

4. Zaks Rodnay: Programming the 6502. Third edition. Sybex, USA, 1980, s. 386

Książki o Atari

5. Atari Intern. Powielone opracowane tłumaczenie. Warszawa 1986, Biuro Usług Komputerowych "Eurobit", s. 201

6. De Re Atari. A Guide to Effective Programming. 400/800 Home Computer. 1981, s. 230

7. Chadwick Ian: Mapping the Atari. COMPUTE! Books, USA, 1985, s. 270

8. COMPUTE!'s First Book of Atari. COMPUTE! Books, 1981, s. 184

9. COMPUTE!'s Third Book of Atari. COMPUTE! Books, 1984, s. 308

Komputery i programowanie

10. Dijkstra Edsger W.: Umiejętność programowania. Wyd.2. Warszawa 1985 WNT, s. 221

11. Myers Glenford J.: Projektowanie niezawodnego oprogramowania. Warszawa 1980 WNT, s. 322

12. Turski Władysław M.: Metodologia programowania. Wyd.2. Warszawa 1982 WNT, s. 264

13. Turski Władysław M.: Propedeutyka informatyki. Wyd.2. Warszawa 1985 WNT, s. 268

14. Wirth Niklaus: Wstęp do programowania systematycznego. Warszawa 1987 WNT, s. 147

15. Gilmore Ch.: Wwiedienije w mikroprocessornuju tiechniku. Moskwa 1984 Mir, s. 334. Przekład książki: Charles M. Gilmore: Introduction to Microprocessors. McGraw Hill 1981

16. Kompiutiery. Sprawocznoje rukowodstwo w trioch tomach. Moskwa 1986 Mir, s. 413,440,406. Tłumaczenie książki: The McGraw Hill Computer Handbook, 1983

17. Sacha Krzysztof, Rydzewski Andrzej: Mikroprocesor w pytaniach i odpowiedziach. Warszawa 1987 WNT, s. 287

18. Ruszczyc Jan: Poznajemy FORTH. Warszawa 1987 SOETO, s. 220.

ANEKSY

A1 Rozkazy 6502. Opis i zastosowania

Poniższe zestawienie obejmuje wszystkie rozkazy mikroprocesora 6502. W nagłówku każdego rozkazu od lewej: mnemonik, jego rozwinięcie angielskie oraz po prawej nazwy znaczników, na które wpływa dany rozkaz. W treści: opis działania rozkazu oraz omówienie najczęstszych zastosowań.

Przy rozkazie BCC obszerniej omówiono wspólne cechy wszystkich ośmiu rozkazów odgałęzień.

Zastosowano następujące oznaczenia:

A - akumulator

X, Y - rejestry indeksowe X i Y

M - bajt w pamięci

DANE, dane - bajt danych w pamięci lub operandzie trybu natychmiastowego.

Nazwy znaczników - takie jak w treści książki (rozdz. 2 i 4).

Dalsze informacje o rozkazach, ich kodach i trybach adresowania zawarte są w następnych aneksach.

<u>ADC</u> add with carry <u>Znaczniki: NVZC</u>

Opis: Dodaje bajt danych do A plus znacznik przeniesienia C. Ustawia C, gdy wynik przekroczył 255 (FF hex). Wynik zostawia w A.

Zastosowania: Jeżeli w chwili dodawania znacznik C był ustawiony (miał wartość 1), wynik będzie o 1 większy od sumy liczb. Dlatego przed rozpoczęciem każdej operacji dodawania należy skasować C z pomocą rozkazu CLC. Jeżeli po wykonaniu ADC znacznik C został ustawiony, oznacza to, że wynik przekroczył rozmiar bajtu czyli 255. W akumulatorze znajduje się wówczas osiem dolnych bitów wyniku, a w C - najbardziej znaczący. Umożliwia to wykorzystanie wartości C w dodawaniu liczb

dwu i wielobajtowych. Patrz punkt 3.5.

ADC wpływa również na inne znaczniki. Ustawienie znacznika nadmiaru V sygnalizuje, że nastąpiło przeniesienie z bitu b6 do b7. V wykorzystywany jest w dodawaniu i odejmowaniu ze znakiem.

Duże znaczenie ma Z, którego ustawienie sygnalizuje wynik 0. Ustawienie N oznacza, że bit b7 wyniku przybrał wartość 1, co w arytmetyce ze znakiem oznacza liczbę ujemną.

Po przejściu z pomocą SED na tryb binarno-dziesiętny rozkaz ADC wykonuje dodawania w tym trybie.

Jako jeden z dwóch rozkazów arytmetycznych ADC jest szeroko wykorzystywany we wszelkich obliczeniach, w tym w podprogramach mnożenia.

AND logical AND Znaczniki: NZ

Opis: Wykonuje logiczne I (koniunkcję) na bitach danych i A. Wynik zostawia w A. Bity wyniku przybierają wartość 1 tylko wtedy, gdy oba odpowiednie bity danych i A mają wartość 1.

Zastosowania: Głównym zastosowaniem AND jest kasowanie części bitów z pomocą wzorcowego bajtu zwanego maską przy zachowaniu wartości pozostałych bitów. Oto przykład:

```
LDA NN      Przetwarzana liczba
AND #$7F    01111111 bin
```

spowoduje, że bit b7 w liczbie NN bez względu na jego poprzednią wartość przybierze 0, natomiast pozostałe bity nie ulegną zmianie.

ASL arithmetic shift left Znaczniki: NZC

Opis: Przesuwa wszystkie bity w bajcie wskazanym przez operand o 1 w lewo i zostawia wynik w tym bajcie. W trybie akumulatora jest to A, w pozostałych trybach M. Skrajny lewy bit bajtu jest przenoszony do znacznika C, a do skrajnego prawego wprowadzane jest 0.

Zastosowania: Umożliwia szybkie mnożenie przez 2. Liczby większe niż 255 można mnożyć przez 2 łącząc stosowanie ASL i ROL. Patrz punkt 3.10. ASL jest szeroko wykorzystywane w pod-

programach mnożenia.

Innym zastosowaniem ASL jest przesuwanie dolnego półbajtu do górnego i wypełnienie dolnego zerami. Uzyskuje się to czterokrotnym ASL.

BCC branch if carry clear Znaczniki: żaden

Opis: Grupa ośmiu rozkazów odgałęzień czyli skoków warunkowych wykonuje czynności na takich samych zasadach, które zostaną tu omówione.

1. Każdy rozkaz odgałęzienia reaguje przeskokiem pod nowy adres w programie na inny z czterech znaczników: N, V, Z i C, w rejestrze stanu procesora P oraz na inną wartość tego znacznika 0 lub 1...Gdy odpowiedni znacznik ma odmienną wartość, przeskok nie następuje i wykonywany jest rozkaz w programie następujący bezpośrednio po rozkazie odgałęzienia.

2. Zawsze w 1-bajtowym operandzie zawarte jest przesunięcie w stosunku do adresu rozkazu następującego bezpośrednio za rozkazem odgałęzienia traktowane jako liczba z e z n a k i e m, dodawana do owego adresu. Tak więc zasięg skoku warunkowego ograniczony jest do 127 bajtów naprzód i 128 bajtów wstecz licząc od następnego rozkazu czyli +129 i -126 licząc od miejsca, w którym umieszczamy rozkaz odgałęzienia.

3. Rozkazy nie wpływają na stan znaczników ani rejestrów.

4. W asemblerze podaje się adres docelowy lub jego etykietę, a program asemblujący sam wylicza właściwy 1-bajtowy operand.

Wracając do rozkazu BCC, wykonuje on skok, gdy znacznik przeniesienia C=0.

Zastosowania: W oparciu o sprawdzenie wyniku CMP, ADC i innych rozkazów wpływających na C rozkaz BCC umożliwia wykonanie odgałęzienia czyli skoku warunkowego analogicznego do struktur IF ... THEN lub ON ... GOTO w Basicu. Przy porównaniach wykonuje odgałęzienie, gdy

DANE > A

BCS branch if carry set Znaczniki: żaden

Opis: Działanie analogiczne jak BCC, ale gdy znacznik

przeniesienia C jest ustawiony (C=1). Gdy C=0, wykonywany jest następny rozkaz.

Zastosowania: Podobne jak BCC. Przy porównaniach wykonuje odgałęzienie, gdy

DANE ⩽ A

BEQ branch if result zero Znaczniki: żaden

Opis: Wykonuje czynności analogiczne jak BCC, gdy znacznik wyniku zerowego Z jest ustawiony Z=1 . Gdy Z=0, wykonywany jest następny rozkaz.

Zastosowania: Umożliwia wykonanie odgałęzienia warunkowego do innego fragmentu programu w przypadku, gdy sprawdzenie Z wykazało równość dwóch liczb lub zerowy wynik operacji. Przy porównaniach wykonuje odgałęzienie, gdy

DANE = A

BIT test bits Znaczniki: NVZ

Opis: Wykonuje logiczne AND na A i M, jednak nie zapisuje wyniku, natomiast wpływa na znaczniki. Znacznik Z ustawiony jest, gdy wynikiem logicznego AND jest 0, kasowany w przeciwnym wypadku. Ponadto bity b7 i b6 danych są k o p i o w a n e odpowiednio do znaczników N i V.

Zastosowania: Zaletą BIT jest to, że przy sprawdzaniu nie zmienia wartości A ani M. BIT upraszcza sprawdzanie bitów b6 i b7 w danych. Użyteczność rozkazu ogranicza to, że nie można stosować go w trybie natychmiastowym ani w adresowaniu indeksowym. Dlatego zastępuje się go często innymi sprawdzeniami, np. CMP i AND.

BMI branch if minus Znaczniki: żaden

Opis: Wykonuje czynności analogiczne jak BCC, gdy znacznik wyniku ujemnego N jest ustawiony (N=1). Gdy N=0, wykonywany jest następny rozkaz.

Zastosowania: Pozwala wykonać odgałęzienie, gdy bit b7 jest ustawiony, tzn. bajt ma wartość większą niż 127. Wykorzystuje się to m.in. w przeszukiwaniu tablic słów kluczowych języków programowania w interpretatorach ustawiając b7 w os-

tatnim znaku słowa. Poza tym możliwe do wykorzystania głównie w arytmetyce ze znakiem.

BNE branch on not equal to zero Znaczniki: żaden

Opis: Wykonuje czynności analogiczne jak BCC, gdy znacznik wyniku ujemnego jest skasowany (Z=0). Gdy Z=1, wykonywany jest następny rozkaz.

Zastosowania: Powoduje odgałęzienie tylko wtedy, gdy porównywane liczby są nierówne. Obok zastosowań analogicznych jak w konstrukcjach Basicu IF ... THEN i ON ... GOTO rozkaz BNE jest również szeroko wykorzystywany w budowie pętli liczonych podobnych do konstrukcji Basicu: FOR I=1 TO N; sekwencja instrukcji: NEXT I. Na przykład:

```
     LDX #$20
     LDA #0
CYKL STA $4000,X
     DEX
     BNE CYKL
```

Powrót do etykiety CYKL następować będzie, dopóki w X nie pojawi się 0, czyli 32 ($20) razy.

Przy porównaniach BNE wykonuje odgałęzienie, gdy

DANE <> A

BPL branch if plus Znaczniki: żaden

Opis: Wykonuje czynności analogiczne jak BCC, gdy znacznik wyniku ujemnego jest skasowany (N=0). Gdy N=1 wykonywany jest następny rozkaz.

Zastosowania: Może być użyty do sprawdzania, czy bajty mają najwyższy bit skasowany, reprezentują zatem wartości mniejsze niż 128, co może być istotne przy kodach znaków. Poza tym możliwe do wykorzystania głównie w arytmetyce ze znakiem.

BRK break Znaczniki: B

Opis: Wywołuje przerwanie programowe. Ustawia znacznik B, po czym wstawia kolejno na stos licznik programu (mniej zna-

czący bajt jako pierwszy) i rejestr stanu procesora P. W PCL i PCH umieszczone zostają zawartości komórek o adresach odpowiednio FFFE i FFFF. W przeciwieństwie do innych przerwań BRK zapisuje stan PC zwiększony o 2, choć jest rozkazem 1-bajtowym.

Zastosowania: BRK wykorzystywane jest przede wszystkim przy uruchamianiu programów w JM w celu ustanawiania punktów przerwań i testowania poprzedzających je odcinków programów.

Klawisz BREAK wywołuje wykonanie BRK. Rozkaz ten powoduje czynności identyczne jak instrukcja STOP w Basicu.

BVC branch if overflow clear Znaczniki: żaden

Opis: Wykonuje czynności analogiczne jak BCC, gdy znacznik nadmiaru V jest skasowany (V=0). Gdy V=1, wykonywany jest następny rozkaz.

Zastosowania: Tylko w arytmetyce ze znakiem. Umożliwia korygowanie wyników w przypadku nadmiaru.

BVS branch if overflow set Znaczniki: żaden

Opis: Wykonuje czynności jak BCC, gdy znacznik V jest ustawiony (V=1). W przeciwnym wypadku wykonywany jest następny rozkaz.

Zastosowania: Jak BVC.

CLC clear carry Znaczniki: C

Opis: Kasuje znacznik przeniesienia C (C=0) w rejestrze P.

Zastosowania: Niezbędny jest przed każdą operacją dodawania. W przypadku dodawania liczb wielobajtowych należy zastosować CLC tylko przed, wykonywanym jako pierwsze, dodaniem najmniej znaczących bajtów.

6502 nie ma rozkazu dodawania bez przeniesienia, stąd konieczność stosowania CLC przed ADC.

CLD clear decimal mode Znaczniki: D

Opis: Kasuje znacznik trybu binarno-dziesiętnego BCD w rejestrze P.wprowadzając tryb binarny dla wszystkich rozkazów ADC i SBC.

Zastosowania: Gdy stosujemy arytmetykę liczb binarnych, celowe jest wprowadzenie tego rozkazu raz na początku programu, by zapobiec przypadkowemu przejściu 6502 na tryb binarno-dziesiętny BCD i dezorganizacji obliczeń. Basic używa kodu BCD w reprezentowaniu liczb zmiennopozycyjnych.

CLI clear interrupt disable Znaczniki: I

Opis: Kasuje znacznik zakazu przerwań I w rejestrze P. Oznacza to zezwolenie na wszelkie przerwania łącznie z maskowalnymi.

Zastosowania: CLI służy do przywrócenia normalnego trybu obsługi przerwań po czasowym jego wyłączeniu z pomocą SEI.

CLV clear overflow Znaczniki: V

Opis: kasuje znacznik nadmiaru V (V=0) w rejestrze P.

Zastosowania: W arytmetyce ze znakiem.

CMP compare data and accumulator Znaczniki: NZC

Opis: Odejmuje dane od akumulatora (A-DANE), ale nie zapisuje wyniku, wpływa natomiast na znaczniki N, Z i C odpowiednio do tego, czy wynik jest dodatni, równy zeru lub ujemny. Wartość A pozostaje niezmieniona.

Z jest ustawiany, gdy oba bajty są sobie równe, kasowany w przeciwnym przypadku. N przybiera wartość bitu b7 wyniku, C jest ustawiany, gdy A $\geqslant$ DANE, kasowany, gdy A $<$ DANE.

Zastosowania: Jest to ważny rozkaz, odgrywający kluczową rolę w konstrukcjach typu: IF ... THEN, ON ... GOTO i FOR ... NEXT. Wraz z następującymi po nim rozkazami odgałęzień CMP umożliwia wybór alternatywnych dróg dalszego wykonywania programu, zależnie od wyniku porównania.

Często poprzednie czynności wpływają na znaczniki w sposób umożliwiający zastosowanie rozkazu odgałęzienia bez CMP. Na przykład, rozkaz LDA #20 skasuje znaczniki N i Z, toteż zastosowanie BPL CEL zawsze spowoduje przeskok do CEL.

Jednakże gdy warunkiem odgałęzienia jest określona wartość niezerowa danych, można to ustalić tylko z pomocą CMP. Rozkazy odgałęzień następująco reagują skokiem na poszczegól-

ne relacje danych i zawartości akumulatora:

DANE > A	BCC
DANE ≤ A	BCS
DANE = A	BEQ
DANE <> A	BNE

Ograniczoną rolę po CMP spełniają BPL i BMI, ponieważ stan bitu b7 nie zawsze prawidłowo określa relację danych i A.

CPX　　compare data and X　　Znaczniki: NZC

Opis: Odejmuje dane od rejestru X (X-DANE), ale nie zapisuje wyniku, wpływa natomiast na znaczniki N, Z i C, odpowiednio do wyniku porównania. Zawartość X pozostaje niezmieniona.

Zastosowania: CPX można stosować zamiennie z CMP wobec identycznego działania obu rozkazów. Można zatem wykorzystać informacje podane przy CMP zastępując oznaczenia A przez X.

Ponieważ rejestr X jest często wykorzystywany do sterowania pętlami i organizacji tablic, w tych dziedzinach CPX odgrywa znaczną rolę. Rozkaz ten, podobnie jak CMP, może być często pominięty, gdy poprzednie czynności określają wartości znaczników. Np. kolejne DEX w pętli doprowadzają w końcu do wyzerowania X i zakończenia działania BNE bez potrzeby stosowania CPX.

CPY　　compare data and Y　　Znaczniki: NZC

Opis: Odejmuje dane od rejestru Y (Y-DANE), ale nie zapisuje wyniku, wpływa natomiast na znaczniki N, Z i C odpowiednio do wyniku porównania. Zawartość Y pozostaje niezmieniona.

Zastosowania: Takie same jak CPX. Rejestr Y jest częściej stosowany w wygodnych trybach adresowania pośredniego indeksowanego i wówczas CPY znajduje często zastosowanie.

DEC　　decrement memory　　Znaczniki: NZ

Opis: Zmniejsza o 1 wartość bajtu pamięci zostawiając wynik w tym bajcie M=M-1 . Wpływa na znaczniki N i Z. N i e

wpływa na znacznik przeniesienia C.

Zastosowania: Użytecznie zastępuje SBC przy odejmowaniu niewielkich liczb. Na przykład (obok to samo z SBC):

```
DEC 2000            LDA 2000
DEC 2000            SEC
                    SBC #2
                    STA 2000
```

Pierwsze rozwiązanie jest krótsze i szybsze.

Z pomocą DEC i komórki pamięci, zwłaszcza na stronie zerowej, można zorganizować licznik pętli, gdy rejestry X i Y są przeciążone.

DEX decrement X register Znaczniki: NZ

Opis: Zmniejsza o 1 zawartość rejestru X (X=X-1). Nie wpływa na C.

Zastosowania: W organizowaniu pętli z pomocą rejestru X. Na przykład:

```
     LDX #20
CYKL  sekwencja rozkazów
     DEX
     BNE CYKL
```

DEY decrement Y register Znaczniki: NZ

Opis: Zmniejsza o 1 zawartość rejestru Y (Y=Y-1). Nie wpływa na C.

Zastosowania: Takie jak DEX, przy czym rejestr Y jest częściej stosowany ze względu na bardzo popularny tryb adresowania pośredniego postindeksowanego Y.

EOR exclusive OR Znaczniki: NZ

Opis: Wykonuje logiczne ALBO (nierównoważność) na bitach danych i A. Bity wyniku przybierają wartość 1 tylko wtedy, gdy odpowiednie bity danych i A mają r ó ż n e wartości. 1 EOR 1 i 0 EOR 0 dają 0.

Zastosowania: Jednym z nich jest powodowanie zmiany wartości najwyższego bitu w kodzie znaku, co w Atari powoduje in-

wartowanie jego obrazu (zamianę barw znaku i tła). EOR jest użyteczne w technikach wyszukiwania z pomocą map bitowych.

INC increment memory Znaczniki: NZ

Opis: Zwiększa o 1 wartość bajtu w pamięci (M=M+1).

Zastosowania: Podobne jak DEC z tą różnicą, że odlicza w górę.

INX increment X register Znaczniki: NZ

Opis: Zwiększa o 1 zawartość rejestru X (X=X+1).

Zastosowania: Podobne jak DEX, z tym, że odlicza w górę.

INY increment Y register Znaczniki: NZ

Opis: Zwiększa o 1 zawartość rejestru Y (Y=Y+1).

Zastosowania: Podobne jak DEY z tym, że odlicza w górę.

JMP jump to new location Znaczniki: żaden

Opis: Wykonuje skok bezwarunkowy w programie pod nowy adres. Nadaje licznikowi programu PC wartość podaną bezpośrednio lub pośrednio w operandzie. Jedyny rozkaz 6502 dostępny w trybie nieindeksowanym.

Zastosowania: Wykonywanie dalekich skoków w programie. Spełnia funkcje analogiczne jak GOTO w Basicu.

Skok pośredni może być użyteczny w przypadku wykorzystywania adresów systemu operacyjnego dostępnych jako wektory w określonych komórkach pamięci. W Atari jednym z wielu tego przykładów jest adres uruchomienia programu umieszczany w komórkach 2E0-2E1.

JSR jump to subroutine Znaczniki: żaden

Opis: Wykonuje skok do podprogramu pod dowolny adres w pamięci. W celu zapewnienia powrotu z podprogramu adres następnego po JSR rozkazu pomniejszony dla uproszczenia pracy 6502 o 1 zostaje wstawiony na stos. Kończący podprogram obowiązkowy rozkaz RTS zdejmuje ten adres ze stosu i zwiększa o 1 , co zapewnia kontynuację programu.

Zastosowania: Jako bezpośredni równoważnik GOSUB Basicu

rozkaz JSR znajduje szerokie zastosowanie, gdy sekwencja rozkazów może być wielokrotnie użyta w programie. JSR pozwala także wykorzystywać podprogramy systemu operacyjnego, np. wykonujące wyświetlanie znaków oraz ich odczytywanie z klawiatury.

Jeżeli po zakończeniu podprogramu chcemy wykonać skok do innego miejsca w programie, należy pamiętać o zdjęciu ze stosu adresu umieszczonego tam przez JSR z pomocą PLA, PLA . W przeciwnym wypadku można łatwo przepełnić stos i spowodować załamanie się programu.

LDA load accumulator Znaczniki: NZ

Opis: Ładuje czyli wpisuje bajt danych do akumulatora.

Zastosowania: LDA jest jednym z najszerzej stosowanych rozkazów przesłania danych. Wynika to z roli akumulatora jako najważniejszego rejestru CPU, za którego pośrednictwem następuje na ogół kierowanie danych do ALU i odbiór wyników.

Duża rola LDA wynika z braku możliwości bezpośredniego przesłania danych pamięć-pamięć i konieczności posłużenia się pośrednictwem akumulatora, jakkolwiek dzięki symetrii rozkazów można do tego wykorzystać również rejestry X i Y.

LDX load X register Znaczniki: NZ

Opis: Ładuje czyli wpisuje bajt danych do rejestru X.

Zastosowania: Podobne jak w przypadku LDA, ponieważ X może spełniać część funkcji akumulatora, a jednocześnie jest przydatny jako licznik pętli i indeks tablic.

LDY load Y register Znaczniki: NZ

Opis: Ładuje czyli wpisuje bajt danych do rejestru Y.

Zastosowania: Analogiczne jak LDX.

LSR logical shift right Znaczniki: NZC

Opis: Przesuwa o jeden w prawo wszystkie bity w bajcie pamięci i zostawia w nim wynik. Skrajny prawy bit przenoszony jest do znacznika przeniesienia C, a do skrajnego lewego bitu wpisywane jest 0. W trybie akumulatora jest to A, w pozostałych M.

Zastosowania: Umożliwia dzielenie wartości bajtu przez 2, a w powiązaniu z ROR także dzielenie przez 2 większych liczb. Szeroko wykorzystywane jest w podprogramach dzielenia.

Przesuwanie bitów stosuje się także do wielu innych celów, np. liczenia bitów jedynkowych. Czterokrotne LSR nadaje bajtowi wartość jego górnej połowy.

NOP — no operation — Znaczniki: żaden

Opis: Nie wykonuje żadnego działania.

Zastosowania: Użyteczne przy poprawianiu programów, pozwala usunąć zbędny rozkaz lub fragment programu i zastąpić go odpowiednią liczbą NOP. Przy uruchamianiu pozwala np. czasowo wyeliminować rozkaz JSR i zbadać zachowanie programu po wyłączeniu podprogramu. NOP trwa dwa cykle i może być użyty do tworzenia pętli czasowych, jednak wygodniejsze jest posłużenie się w tym celu zegarami komputera.

ORA — logical OR — Znaczniki: NZ

Opis: Wykonuje logiczne LUB (alternatywę) na bitach danych i akumulatora. Bity wyniku przybierają wartość 1, gdy co najmniej jeden z dwóch odpowiednich bitów danych i A ma wartość 1. Tak więc 0 w wyniku powstaje tylko przy zbiegu dwóch 0.

Zastosowania: Podczas gdy maska AND pozwala kasować bity, to maska OR służy głównie do ustawiania bitów. Jeżeli np. w Atari chcemy, by wszystkie znaki były wyświetlane jako inwertowane bez względu na to, jakie były poprzednio, powinniśmy w ich kodach najwyższy bit ustawić na 1. Wykona to ORA #80. ORA znajduje wiele zastosowań.

PHA — push accumulator on stack — Znaczniki: żaden

Opis: Wstawia A na stos. Odejmuje 1 od wskaźnika stosu S. Zawartość A pozostaje niezmieniona.

Zastosowania: Główne zastosowanie wiąże się z wykorzystywaniem stosu jako miejsca czasowego przechowywania danych. Jest to forma znacznie dogodniejsza niż przechowywanie danych w zmiennej tymczasowej z pomocą STA TEMP. Oszczędza w programie dwa bajty, a gdy TEMP nie jest na stronie zerowej - także

cykl zegarowy. Pamiętać jednak należy o konieczności zdjęcia w porę ze stosu przechowywanych danych, jak i o tym, że JSR "nakrywa" je adresem powrotu i przed odpowiednim RTS nie są dostępne.

Z pomocą PHA często wstawiamy na stos dane z rejestrów w celu ich czasowego przechowania.

PHP push P register on stack Znaczniki: żaden

Opis: Wstawia na stos zawartość rejestru stanu procesora P. Odejmuje 1 od wskaźnika stosu S. Zawartość P pozostaje niezmieniona.

Zastosowania: W niektórych sytuacjach, w tym przy przerwaniach, poprawna kontynuacja programu może zależeć od dokładnego odtworzenia stanu wszystkich znaczników. Rozkaz PHP pozwala czasowo przechować zawierający je rejestr na stosie.

PLA pull accumulator from stack Znaczniki: NZ

Opis: Zdejmuje wartość ze szczytu stosu do akumulatora. Zwiększa o 1 wskaźnik stosu S.

Zastosowania: Pozwala odtworzyć wartość czasowo przechowywaną z pomocą PHA. W Atari PLA stosowane jest również w podprogramach w języku maszynowym wywoływanych z Basicu w celu przyjęcia przekazanych argumentów. Patrz rozdz. 7.

PLP pull P register from stack Znaczniki: wszystkie

Opis: Zdejmuje wartość ze szczytu stosu do rejestru P. Zwiększa o 1 wskaźnik stosu S.

Zastosowania: Pozwala odtworzyć poprzedni stan znaczników przechowywany z pomocą PHP. Wpływa tym samym na wszystkie znaczniki.

ROL rotate one bit left Znaczniki: NZC

Opis: Wykonuje obrót bitów o jedną pozycję w lewo w bajcie danych i zostawia w nim wynik. Obrót - to cykliczne przesunięcie 9 bitów z udziałem znacznika C. Różni się ono od ASL tylko tym, że po przesunięciu bitów do skrajnego prawego opróżniającego się bitu wstawiana jest poprzednia wartość C, a

nie 0. W trybie akumulatora ROL wykonuje obrót bitów w A.

Zastosowania: Wykorzystywane łącznie z ASL w podprogramach mnożenia liczb dwu lub wielobajtowych.

ROR rotate one bit right Znaczniki: NZC

Opis: Wykonuje obrót bitów o jedną pozycję w prawo w bajcie danych i zostawia w nim wynik. Obrót - to cykliczne przesunięcie 9 bitów z udziałem znacznika C. Różni się od LSR tylko tym, że po przesunięciu bitów do opróżniającego się skrajnego lewego wstawiana jest poprzednia wartość C. W trybie akumulatora ROR wykonuje obrót bitów w A.

Zastosowania: Wykorzystywane łącznie z LSR w podprogramach dzielenia liczb dwu i wielobajtowych.

RTI return from interrupt Znaczniki: wszystkie

Opis: Wykonuje powrót z programu obsługi przerwania do wykonywanego programu. RTI przenosi ze stosu do rejestru P oraz do licznika programu ich wartości przechowane przed wywołaniem przerwania. Wpływa zatem na wszystkie znaczniki. Koryguje wskaźnik stosu S zwiększając go łącznie o 3 (1 bajt P i 2 bajty PC). W przeciwieństwie do RTS nie dodaje 1 do zdjętego ze stosu adresu powrotu.

Zastosowania: W programach wykorzystujących mechanizm przerwań.

RTS return from subroutine Znaczniki: żaden

Opis: Zapewnia powrót z podprogramu. Odtwarza ze stosu zwiększając o 1 przechowany automatycznie przez JSR licznik programu. Koryguje wskaźnik stosu S.

Zastosowania: W parze z JSR stanowi szeroko wykorzystywany mechanizm tworzenia podprogramów.

SBC substract with borrow Znaczniki: NVZC

Opis: Odejmuje bajt danych od akumulatora oraz w razie potrzeby pożyczkę z wyższego bajtu. Wynik zostawia w akumulatorze.

Zastosowania: Gdy zachodzi potrzeba pożyczki znacznik

przeniesienia jest kasowany (C=0), co jest jakby odwróceniem funkcji C pełnionej przy dodawaniu (wyjaśnienie w punkcie 3.8). Dlatego też odwrotnie niż przy dodawaniu, przed każdą operacją odejmowania należy ustawić znacznik C z pomocą SEC. Wykonując odejmowanie SBC od różnicy odejmuje zaprzeczenie C czyli 1, gdy C=0 i 0, gdy C=1.

Jeżeli wykonuje się odejmowanie liczb dwu lub wielobajtowych, C należy ustawić z pomocą SEC tylko przed odejmowaniem najmniej znaczących bajtów.

SBC wpływa również na inne znaczniki. V wykorzystywany jest przy odejmowaniu liczb ze znakiem do ewentualnej korekty znaku. N=1 oznacza, że bit znaku w wyniki przybrał wartość 1.

Ustawienie Z (Z=1) sygnalizuje, że powstał wynik 0.

Po przejściu z pomocą SED na tryb binarno-dziesiętny SBC wykonuje odejmowanie w tym trybie.

Jako jeden z dwóch rozkazów arytmetycznych SBC jest szeroko wykorzystywany we wszelkich obliczeniach, a w szczególności w podprogramach dzielenia.

SEC set carry Znaczniki: C

Opis: Ustawia znacznik przeniesienia C w rejestrze P.

Zastosowania: Przed każdą operacją odejmowania. Patrz SBC.

SED set decimal mode Znaczniki: D

Opis: Ustawia znacznik D w rejestrze stanu procesora P. Powoduje, że dodawania i odejmowania wykonywane są odtąd w kodzie binarno-dziesiętnym BCD .

Zastosowania: Rozkaz ten pozwala przejść na wykonywanie obliczeń w kodzie binarno-dziesiętnym, który jest niekiedy wygodniejszy, np. w obliczeniach finansowych i ekonomicznych, powoduje jednak większe zużycie pamięci i zmniejsza szybkość obliczeń. Patrz punkt 3.11.

SEI set interrupt disable Znaczniki: I

Opis: Ustawia znacznik zakazu przerwań w rejestrze P. Powoduje to, że 6502 wyłącza system przerwań poza nieliczny-

mi przerwaniami niemaskowalnymi, takimi jak gorący i zimny start.

Zastosowania: Umożliwia zmianę znajdującego się w RAM wektora do programu obsługi przerwań, tak aby wskazywał adres naszego własnego programu. Można to jednak wykonać tylko po wyłączeniu innych przerwań z pomocą SEI, a po zmianie wektora (LDA STA LDA STA) należy ponownie uruchomić przerwania z pomocą CLI. Atari zapewnia także inne sposoby wykorzystania przerwań przez programującego.

STA store accumulator in memory Znaczniki: żaden

Opis: Zapisuje A we wskazanej komórce pamięci. Nie zmienia zawartości A.

Zastosowania: Jeden z najczęściej stosowanych rozkazów. Ponieważ w akumulatorze ALU zostawia wyniki większości obliczeń (po ADC, SBC, CMP i wielu innych rozkazach), STA służy do zapisania tych wyników w pamięci.

Wraz z LDA umożliwia przesłania danych pamięć-pamięć z wykorzystaniem pośrednictwa akumulatora. 6502 nie ma rozkazów pozwalających na bezpośrednie przesłanie pamięć-pamięć.

STX store X in memory Znaczniki: żaden

Opis: Zapisuje zawartość X we wskazanej komórce pamięci. Nie zmienia X.

Zastosowania: Podobne jak STA z uwzględnieniem odrębnych funkcji i możliwości rejestru X. Identyczne możliwości przesłań pamięć-pamięć.

STY store Y register in memory Znaczniki: żaden

Opis: Zapisuje zawartość Y we wskazanej komórce pamięci. Nie zmienia Y.

Zastosowania: Jak STA i STX.

TAX transfer accumulator into X Znaczniki: NZ

Opis: Przesyła zawartość A do X. Nie zmienia A.

Zastosowania: Jeden z sześciu rozkazów przesłań między rejestrami wewnętrznymi 6502, których znaczenie łatwo jest od-

czytać z mnemoników: po pierwszej literze "T" dwie następne wskazują rejestry, a ich kolejność - kierunek przesłania.

TAX pozwala czasowo przechować zawartość akumulatora w rejstrze X, a późniejsze TXA odtworzyć ję w A. Gdy X wykorzystujemy jako indeks tablicy, możliwe jest przesłanie z A wyników obliczeń na nim niemożliwych do wykonania w X.

TYA transfer accumulator into Y Znaczniki: NZ

Opis: Przesyła zawartość A do Y. Nie zmienia A.

Zastosowania: Podobne jak TAX.

TSX transfer stack pointer into X Znaczniki: NZ

Opis: Przesyła wskaźnik stosu S do rejestru X. Nie zmienia wartości S.

Zastosowania: TSX i TXS są jedynymi rozkazami umożliwiającymi w 6502 dwustronną komunikację ze wskaźnikiem stosu S. W przypadku przesłań ze stosu i na stos S jest automatycznie zmieniany i zawsze wskazuje pierwszą wolną komórkę na stosie. Gdy stos rośnie, wartość S maleje i na odwrót, co jest następstwem faktu, iż umieszczony jest w pamięci "do góry nogami", szczytem ku dołowi.

TSX pozwala odczytać, a TXS ewentualnie zmienić wartość wskaźnika stosu, co jest jednak operacją trudną i wymaga bardzo starannego wykonania. Stos pracuje najlepiej wtedy, gdy pozwolimy, by "rządził" nim sam mikroprocesor.

TXA transfer X into accumulator Znaczniki: NZ

Opis: Przesyła zawartość X do A. Nie zmienia X.

Zastosowania: Gdy wykorzystujemy X jako wskaźnik tablicy, celowe jest nieraz przeniesienie go do A, by np. po znalezieniu poszukiwanego znaku dodać wskaźnik do początkowego adresu tablicy i uzyskać adres znaku w pamięci. W X nie można wykonać dodawania.

Poza tym TXA znajduje zastosowanie omówione przy TAX.

TXS transfer X into stack pointer Znaczniki: żaden

Opis: Przesyła zawartość X do S. Nie zmienia X.

Zastosowania: Omówione przy TSX.

TYA   transfer Y into accumulator   Znaczniki: NZ

Opis: Przesyła Y do A. Nie zmienia Y.
Zastosowania: Jak TXA.

A2 Rozkazy i kody operacji alfabetycznie

W tabeli kolejno od lewej: mnemonik i forma zapisu trybu, kod operacji w hex i dec, długość w bajtach i liczba cykli zegarowych.

Oznaczenia: #n - dane w trybie natychmiastowym, Q - adres dwubajtowy, Z - adres 1-bajtowy danych lub adresu pośredniego, L - przesunięcie ze znakiem, A, X i Y - rejestry 6502.

+ - dodać cykl przy zmianie strony pamięci, ! - odjąć cykl, gdy nie ma skoku, dodać cykl, gdy skok na inną stronę.

Asembler	Kod op. hex	dec	Dł.	Cykl.	Asembler	Kod op. hex	dec	Dł.	Cykl.
ADC #n	69	105	2	2	BPL L	10	16	2	3!
ADC Q	6D	109	3	4	BRK	00	0	1	7
ADC Z	65	101	2	3	BVC L	50	80	2	3!
ADC Q,X	7D	125	3	4+	BVS L	70	112	2	3!
ADC Q,Y	79	121	3	4+	CLC	18	24	1	2
ADC Z,X	75	117	2	4	CLD	D8	216	1	2
ADC (Z,X)	61	97	2	6	CLI	58	88	1	2
ADC (Z),Y	71	113	2	5+	CLV	B8	184	1	2
AND #n	29	41	2	2	CMP #n	C9	201	2	2
AND Q	2D	45	3	4	CMP Q	CD	205	3	4
AND Z	25	37	2	3	CMP Z	C5	197	2	3
AND Q,X	3D	61	3	4+	CMP Q,X	DD	221	3	4+
AND Q,Y	39	57	3	4+	CMP Q,Y	D9	217	3	4+
AND Z,X	35	53	2	4	CMP Z,X	D5	213	2	4
AND (Z,X)	21	33	2	6	CMP (Z,X)	C1	193	2	6
AND (Z),Y	31	49	2	5+	CMP (Z),Y	D1	209	2	5+
ASL A	0A	10	1	2	CPX #n	E0	224	2	2
ASL Q	0E	14	2	6	CPX Q	EC	236	3	4
ASL Z	06	6	2	5	CPX Z	E4	228	2	3
ASL Q,X	1E	30	3	7	CPY #n	C0	192	2	2
ASL Z,X	16	22	2	6	CPY Q	CC	204	3	4
BCC L	90	144	2	3!	CPY Z	C4	196	2	3
BCS L	B0	176	2	3!	DEC Q	CE	216	3	6
BEQ L	F0	240	2	3!	DEC Z	C6	198	2	5
BIT Q	2C	44	3	4	DEC Q,X	DE	222	3	7
BIT Z	24	36	2	3	DEC Z,X	D6	214	2	5
BMI L	30	48	2	3!	DEX	CA	202	1	2
BNE L	D0	208	2	3!	DEY	88	136	1	2

Asembler	Kod op. hex	Kod op. dec	Dł.	Cykl.
EOR #n	49	73	2	2
EOR Q	4D	77	3	4
EOR Z	45	69	2	3
EOR Q,X	5D	93	3	4+
EOR Q,Y	59	89	3	4+
EOR Z,X	55	85	2	4
EOR (Z,X)	41	65	2	6
EOR (Z), Y	51	81	2	5+
INC Q	EE	238	3	6
INC Z	E6	230	2	5
INC Q,X	FE	254	3	7
INC Z,X	F6	246	2	6
INX	E8	232	1	2
INY	C8	200	1	2
JMP Q	4C	76	3	3
JMP (Q)	6C	108	3	5
JSR Q	20	32	3	6
LDA #n	A9	169	2	2
LDA Q	AD	173	3	4
LDA Z	A5	165	2	3
LDA Q,X	BD	189	3	4+
LDA Q,Y	B9	185	3	4+
LDA Z,X	B5	181	2	4
LDA (Z,X)	A1	161	2	6
LDA (Z),Y	B1	177	2	5+
LDX #n	A2	162	2	2
LDX Q	AE	174	3	4
LDX Z	A6	166	2	3
LDX Q,Y	BE	190	3	4+
LDX Z,Y	B6	182	2	4
LDY #n	A0	160	2	2
LDY Q	AC	172	3	4
LDY Z	A4	164	2	3
LDY Q,X	BC	188	3	4+
LDY Z,X	B4	180	2	4
LSR A	4A	74	1	2
LSR Q	4E	78	3	6
LSR Z	46	70	2	5
LSR Q,X	5E	94	3	7
LSR Z,X	56	86	2	6
NOP	EA	234	1	2
ORA #n	09	9	2	2
ORA Q	0D	13	3	4
ORA Z	05	5	2	3
ORA Q,X	1D	29	3	4+
ORA Q,Y	19	25	3	4+
ORA Z,X	15	21	2	4
ORA (Z,X)	01	1	2	6
ORA (Z),Y	11	17	2	5+
PHA	48	72	1	3
PHP	08	8	1	3
PLA	68	104	1	4
PLP	28	40	1	4
ROL A	2A	42	1	2
ROL Q	2E	46	3	6
ROL Z	26	38	2	5
ROL Q,X	3E	62	3	7
ROL Z,X	36	54	2	6
ROR A	6A	106	1	2
ROR Q	6E	110	3	6
ROR Z	66	102	2	5
ROR Q,X	7E	126	3	7
ROR Z,X	76	118	2	6
RTI	40	64	1	6
RTS	60	96	1	6
SBC #n	E9	233	2	2
SBC Q	ED	237	3	4
SBC Z	E5	229	2	3
SBC Q,X	FD	253	3	4+
SBC Q,Y	F9	249	3	4+
SBC Z,X	F5	245	2	4
SBC (Z,X)	E1	225	2	6
SBC (Z),Y	F1	241	2	5+
SEC	38	56	1	2
SED	F8	248	1	2
SEI	78	120	1	2
STA Q	8D	141	3	4
STA Z	85	133	2	3
STA Q,X	9D	157	3	5
STA Q,Y	99	153	3	5
STA Z,X	95	149	2	4
STA (Z,X)	81	129	2	6
STA (Z),Y	91	145	2	6
STX Q	8E	142	3	4
STX Z	86	134	2	3
STX Z,Y	96	150	2	4
STY Q	8C	140	3	4
STY Z	84	132	2	3
STY Z,X	94	148	2	4
TAX	AA	170	1	2
TAY	A8	168	1	2
TSX	BA	186	1	2
TXA	8A	138	1	
TXS	9A	154	1	2
TYA	98	152	1	2

<u>Uwaga</u>: Forma zapisu trybów adresowania omówiona została w punkcie 5.1.

A3 Rozkazy w kolejności rosnących kodów operacji
Oznaczenia jak w poprzednim aneksie.

Kod hex	Kod dec	Asembler	Kod hex	Kod dec	Asembler	Kod hex	Kod dec	Asembler
00	0	BRK	35	53	AND Z,X	6A	106	ROR A
01	1	ORA (Z,X)	36	54	ROL Z,X	6C	108	JMP (Q)
05	5	ORA Z	38	56	SEC	6D	109	ADC Q
06	6	ASL Z	39	57	AND Q,Y	6E	110	ROR Q
08	8	PHP	3D	61	AND Q,X	70	112	BVS L
09	9	ORA #n	3E	62	ROL Q,X	71	113	ADC (Z),Y
0A	10	ASL A	40	64	RTI	75	117	ADC Z,X
0D	13	ORA Q	41	65	EOR (Z,X)	76	118	ROR Z,X
0E	14	ASL Q	45	69	EOR Z	78	120	SEI
10	16	BPL L	46	70	LSR Z	79	121	ADC Q,Y
11	17	ORA (Z),Y	48	72	PHA	7D	125	ADC Q,X
15	21	ORA Z,X	49	73	EOR #n	7E	126	ROR Q,X
16	22	ASL Z,X	4A	74	LSR A	81	129	STA (Z,X)
18	24	CLC	4C	76	JMP Q	84	132	STY Z
19	25	ORA Q,Y	4D	77	EOR Q	85	133	STA Z
1D	29	ORA Q,X	4E	78	LSR Q	86	134	STX Z
1E	30	ASL Q,X	50	80	BVC L	88	136	DEY
20	32	JSR Q	51	81	EOR (Z),Y	8A	138	TXA
21	33	AND (Z,X)	55	85	EOR Z,X	8C	140	STY Q
24	36	BIT Z	56	86	LSR Z,X	8D	141	STA Q
25	37	AND Z	58	88	CLI	8E	142	STX Q
26	38	ROL Z	59	89	EOR Q,Y	90	144	BCC L
28	40	PLP	5D	93	EOR Q,X	91	145	STA (Z),Y
29	41	AND #n	5E	94	LSR Q,X	94	148	STY Z,X
2A	42	ROL A	60	96	RTS	95	149	STA Z,X
2C	44	BIT Q	61	97	ADC (Z,X)	96	150	STX Z,Y
2D	45	AND Q	65	101	ADC Z	98	152	TYA
2E	46	ROL Q	66	102	ROR Z	99	153	STA Q,Y
30	48	BMI L	68	104	PLA	9A	154	TXS
31	49	AND (Z),Y	69	105	ADC #n	9D	157	STA Q,X

Kod hex	dec	Asembler	Kod hex	dec	Asembler	Kod hex	dec	Asembler
A0	160	LDY #n	BD	189	LDA Q,X	DE	222	DEC Q,X
A1	161	LDA (Z,X)	BE	190	LDX Q,Y	E0	224	CPX #n
A2	162	LDX #n	C0	192	CPY #n	E1	225	SBC (Z,X)
A4	164	LDY Z	C1	193	CMP (Z,X)	E4	228	CPX Z
A5	165	LDA Z	C4	196	CPY Z	E5	229	SBC Z
A6	166	LDX Z	C5	197	CMP Z	E6	230	INC Z
A8	168	TAY	C6	198	DEC Z	E8	232	INX
A9	169	LDA #n	C8	200	INY	E9	233	SBC #n
AA	170	TAX	C9	201	CMP #n	EA	234	NOP
AC	172	LDY Q	CA	202	DEX	EC	236	CPX Q
AD	173	LDA Q	CC	204	CPY Q	ED	237	SBC Q
AE	174	LDX Q	CD	205	CMP Q	EE	238	INC Q
B0	176	BCS L	CE	206	DEC Q	F0	240	BEQ L
B1	177	LDA (Z),Y	D0	208	BNE L	F1	241	SBC (Z),Y
B4	180	LDY Z,X	D1	209	CMP (Z),Y	F5	245	SBC Z,X
B5	181	LDA Z,X	D5	213	CMP Z,X	F6	246	INC Z,X
B6	182	LDX Z,X	D6	214	DEC Z,X	F8	248	SED
B8	184	CLV	D8	216	CLD	F9	249	SBC Q,Y
B9	185	LDA Q,Y	D9	217	CMP Q,Y	FD	253	SBC Q,X
BA	186	TSX	DD	221	CMP Q,X	FE	254	INC Q,X
BC	188	LDY Q,X						

Przykłady stosowania tablic A4 i A5:

Należy sumować wartości odpowiadające kolejnym cyfrom:

Hex na dec: liczba C3AF

Czwarta cyfra	C =	49152
Trzecia cyfra	3 =	768
Druga cyfra	A =	160
Pierwsza cyfra	F =	15
		50095

Dec na hex: liczba 38706

Piąta cyfra	3 =	7530
Czwarta cyfra	8 =	1F40
Trzecia cyfra	7 =	2BC
Pierwsza cyfra	6 =	6
		9732

A4 Przekształcanie hex w dec

	4 cyfra	3 cyfra	2 cyfra	1 cyfra
0	0	0	0	0
1	4096	256	16	1
2	8192	512	32	2
3	12288	768	48	3
4	16384	1024	64	4
5	20480	1280	80	5
6	24576	1536	96	6
7	28672	1792	112	7
8	32768	2048	128	8
9	36864	2304	144	9
A	40960	2560	160	10
B	45056	2816	176	11
C	49152	3072	192	12
D	53248	3328	208	13
E	57344	3584	224	14
F	61440	3840	240	15

A5 Przekształcanie dec w hex

	5 cyfra	4 cyfra	3 cyfra	2 cyfra	1 cyfra
0	0	0	0	0	0
1	2710	3E8	64	A	1
2	4E20	7D0	C8	14	2
3	7530	BB8	12C	1E	3
4	9C40	FA0	190	28	4
5	C350	1338	1F4	32	5
6	EA60	1770	258	3C	6
7	11170	1B58	2BC	46	7
8	13880	1F40	320	50	8
9	16F90	2328	384	5A	9

Przykłady stosowania na poprzedniej stronie.

A6 Grupy kodów operacji 6502

Bajty kodów operacji 6502 składają się z pól określających: grupę, do której należy kod; czynność; tryb adresowania. Oto podział kodów na grupy wraz z wartościami pól. Opis oparto na [3].

m1, m2, m3, m4, m5, m6 - kod czynności; a0, a1,- kod trybu adresowania. Przy każdej grupie podano strukturę bajtu kodu operacji.

Grupa 00-0

7 6 5 4 3	2	1	0
m1	0	0	0

m1 =	0	BRK
	1	PHP
	2	BPL
	3	CLC
	4	JSR
	5	PLP
	6	BMI
	7	SEC
	8	RTI
	9	PHA
	10	BVC
	11	CLI
	12	RTS
	13	PLA
	14	BVS
	15	SEI
	- - -	
	17	DEY
	18	BCC
	19	TYA
	20	LDY #n
	21	TAY
	22	BCS
	23	CLV
	24	CPY #n
	25	INY
	26	BNE
	27	CLD
	28	CPX #n
	29	INX
	30	BEQ
	31	SED

Grupa 00-1

7	6 5	4 3 2	1	0
1	m2	a0	0	0

			Wartości a1
m2 =	0	STY	1,3,5
	1	LDY	0,1,3,5,7
	2	CPY	0,1,3
	3	CPX	0,1,3

Grupa 00-0-1

7	6 5 4 3	2	1	0
0	m3	1	0	0

m3 =	4	BIT Z
	5	BIT Q
	9	JMP Q
	13	JMP (Q)

Grupa 01

7 6 5	4 3 2	1	0
m4	a1	0	1

a1 =	0	(Z,X)
	1	Z
	2	#n
	3	Q
	4	(Z),Y
	5	Z,X
	6	Q,Y
	7	Q,X

Grupa 10-zwykła

7 6 5	4 3 2	1	0
m5	a0	1	0

a0 =	0	#n
	1	Z
	2	A lub specj.
	3	Q
	5	Z,X lub Z,Y
	6	specjalna
	7	Q,X lub Q,Y

			Wartości a0
m5 =	0	ASL	1,2,3,5,7
	1	ROL	1,2,3,5,7
	2	LSR	1,2,3,5,7
	3	ROR	1,2,3,5,7
	4	STX	1,3,5
	5	LDX	0,1,3,5,7
	6	DEC	1,3,5,7
	7	INC	1,3,5,7

Grupa 10-zwykła

7 6 5 4	3	2	1	0
m6	1	0	1	0

m6 =	0	ASL
	2	ROL
	4	LSR
	6	ROR
	8	TXA
	9	TXS
	10	TAX
	11	TSX
	12	DEX
	14	NOP

A7 Tablica skoków względnych wstecz

Skok L dec	Skok L hex	Skok L dec	Skok L hex	Skok L dec	Skok L hex	Skok L dec	Skok L hex	Skok L dec	Skok L hex	Skok L dec	Skok L hex
1	FF	22	EA	43	D5	64	C0	85	AB	106	96
2	FE	23	E9	44	D4	65	BF	86	AA	107	95
3	FD	24	E8	45	D3	66	BE	87	A9	108	94
4	FC	25	E7	46	D2	67	BD	88	A8	109	93
5	FB	26	E6	47	D1	68	BC	89	A7	110	92
6	FA	27	E5	48	D0	69	BB	90	A6	111	91
7	F9	28	E4	49	CF	70	BA	91	A5	112	90
8	F8	29	E3	50	CE	71	B9	92	A4	113	8F
9	F7	30	E2	51	CD	72	B8	93	A3	114	8E
10	F6	31	E1	52	CC	73	B7	94	A2	115	8D
11	F5	32	E0	53	CB	74	B6	95	A1	116	8C
12	F4	33	DF	54	CA	75	B5	96	A0	117	8B
13	F3	34	DE	55	C9	76	B4	97	9F	118	8A
14	F2	35	DD	56	C8	77	B3	98	9E	119	89
15	F1	36	DC	57	C7	78	B2	99	9D	120	88
16	F0	37	DB	58	C6	79	B1	100	9C	121	87
17	EF	38	DA	59	C5	80	B0	101	9B	122	86
18	EE	39	D9	60	C4	81	AF	102	9A	123	85
19	ED	40	D8	61	C3	82	AE	103	99	124	84
20	EC	41	D7	62	C2	83	AD	104	98	125	83
21	EB	42	D6	63	C1	84	AC	105	97	126	82
										127	81
										128	80

Sposób korzystania:

Dla skoków warunkowych wstecz - odnajdujemy w tablicy odpowiednie operandy w hex. Np. w następującym fragmencie programu chcemy obliczyć operand dla skoku BNE pod adres 680:

```
680 LDA 1000,X
683 STA 2000,X
686 INX
687 BNE 680            Jaki operand?
```

Adres n a s t ę p n e g o rozkazu: 689. Skok o 9 wstecz. Z tablicy odczytujemy operand: F7.

A8 Kody znaków sterujących Atari

hex	dec	
1B	27	ESC
1C	28	Kursor w górę
1D	29	Kursor w dół
1E	30	Kursor w lewo
1F	31	Kursor w prawo
7C	125	Oczyść ekran
7D	126	Usuń wstecz
7E	127	Tabulator
9B	155	RETURN
9C	156	Usuń linię
9D	157	Wstaw linię
9E	158	Skasuj tabulację
9F	159	Ustaw tabulację
FD	253	Buczek (CTL 2)
FE	254	Usuń pod kursorem
FF	255	Wstaw znak

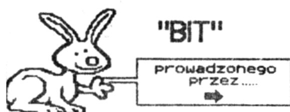
SPIS TREŚCI

- Świetne warunki lokalowe.
- Doskonała lokalizacja w Warszawie.
- Dobre przygotowanie techniczne i organizacyjne.

o tylko niektóre zalety

STUDIA MIKROKOMPUTEROWEGO

"BIT"



STOŁECZNY OŚRODEK ELEKTRONICZNEJ TECHNIKI OBLICZENIOWEJ

SOETO



M "BIT" pełni funkcje:

tudia Dydaktycznego
alonu Gier
racowni Mikroinformatycznej



ORAZ ....

SM "BIT" może pełnic funkcje:

Agencji reklamowej.
Salonu wystawowego.
Pośrednika akwizycyjno-handlowego.
Koordynatora kompletacji sprzetu.
Organizatora pokazów, instruktaży i szkoleń w ramach działalności ,której celem jest upowszechnienie informatyki mikrokomputerowej (hardware , software , systemy mikroinformatyczne)

Polecamy swoje usługi.

v v v v v v v v v v v

NASZ ADRES : SOETO

00-682 WARSZAWA
ul. Hoża 50 ; 4piętro pok.22

telefon : **29-18-64**
telefon : **21-64-01** wew. 65 lub 66

LUB : SOETO-SM„BIT"

00-060 WARSZAWA
ul. Królewska 27 ; wejście B
1piętro pok.205

telefon : **27-72-81** wew. **526**

^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^

Cena zł 910.–





COPYRIGHT BY SOETO

Wydawca SOETO
ul Hoża 50 00–682 Warszawa
tel. 21-64-01 w. 66
tlx 894786

